UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial
Aeroespacial i Audiovisual de Terrassa

Final Degree Report

BACHELOR'S DEGREE IN AEROSPACE VEHICLE ENGINEERING

Study and implementation in C of the Finite Element Method in dimension 2

Eduard Gómez Escandell

Tutor
Rafel Amer Ramon

Escola Superior d'Enginyeries Industrial
Aeroespacial i Audiovisual de Terrassa
June 10, 2019

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial
Aeroespacial i Audiovisual de Terrassa

BACHELOR'S DEGREE IN AEROSPACE VEHICLE ENGINEERING

Study and implementation in C of the Finite Element Method in dimension 2

Eduard Gómez Escandell

Tutor
Rafel Amer Ramon

Escola Superior d'Enginyeries Industrial
Aeroespacial i Audiovisual de Terrassa
June 10, 2019

Eduard Gómez Escandell <eduard.gomez.escandell@gmail.com>

Contents

List of Figures

List of Tables

List of Source Code

CHAPTER 1

Preface

1.1 Abstract

This document explains the finite element method for solving linear partial differential equations in one and two dimensional domains. It consists of seven chapters. This first one acts as a description of the project. The second one explains the problem to be solved. The third chapter explains necessary concepts to understand chapters 5 and 6, where the problem is solved. The fourth chapter is independent, and needs not be read in order to understand later chapters. Finally, the seventh chapter acts as a small conclusion, although the main results are in chapters 5 and 6.

1.2 Aim

The aim of this project is to explore the different mathematical concepts that govern partial differential equations (henceforth PDE) and tools used to solve them with the finite element method (henceforth FEM). It also aims to explore how these tools are translated into code.

By writing the code in a low-level language such as C, this project aims at being explicit about all the steps taken. Higher level languages mean shorter development time but they introduce much more abstraction which could cloud our understanding of the FEM.

It must be made clear that this document does not, and aims not to contain a guide on how to develop a well-polished product. Its content is purely educational: an introduction to the FEM.

1.3 Scope

This project's scope includes:

- Studying the mathematical principles behind the FEM
- Developing a solver for one dimensional domains
- Developing a solver for two dimensional domains
- Reading the domain and problem specifications from an .xml file

For the sake of clarity, I must specify that the following tasks are out of scope:

- Solving time-dependent or non-linear PDE.
- Developing the code to triangulate the two-dimensional domain.

1.4 Specifications

The following were the requirements for the project:

- The programming language must be C.
- The solver will work on one and two dimensional domains.
- The 2d domain must be read from an input file.

1.5 Reason to be

A lot has been written about the finite element method, indeed I have a full stand of books next to me that explain it to the minutest detail. Unfortunately, for the most part they can be classified in two groups: those that are way too technical for a novice engineer, and those that pursuing simplicity become too rigid.

The former analyse every detail of the FEM, and talk about advanced mathematical concepts such as function spaces and other set theory concepts that new engineers are not acquainted with.

The latter simplify the FEM so much that the program, once developed, does not allow for different settings. A good educational tool must allow the learner to toy around with it, to see the effect of, for instance, using lower degree polynomials to interpolate and see how that affects the solution.

The code developed in this project is very open to tweaking and therefore allows the user to see how different parameters affect the solution. The text, on the other hand, does not get more technical than it needs be, so a junior engineer should be more than capable of understanding it.

1.6 State of the art

The finite element method is a very useful tool that can be used in many different fields. A lot has been written about it and many variations exist. Two particularly useful ones are FreeFem++ and the FEniCS project. They are interesting because they are free and open source, and they have plenty of documentation. The FEniCS documentation is so extensive that it even has a free book explaining the FEM to very good detail (see [6] and [7]).

The FEniCS project is developped by Numfocus (of matplotlib and NumPy fame). The core is written in C++ but it distributes packages for C++ and Python. It can solve all sorts of differential equations: linear and non-linear, in all 1 to 3 dimensional domains. It can solve them with different solvers and many options. One can define the domain in the code or import an xml, so it's relatively easy to define the domain in autoCAD or SolidWorks, convert it, and use it with FEniCS. Ultimately it's strongest suit is its simplicity. Programs that use it are acceptably short and straight-forward.

Freefem++ is developed by a long list of (mostly french) developers. Their solver is also written in C++, however to use it one must use their language. The FreeFem++ language is very similar to C++, but not without its own particularities. Just like FEniCS, one can define the domain in the code or import it.

Another interesting piece of software is Simscale. Its particularity is that it is exclusively browser-based. This means that the computing is done off-site (i.e. cloud computing). At its core it uses OpenFoam to solve the partial differential equations, another open-source solver. Simscale adds the pre- and post-processing.

On the proprietary side, and perhaps the most famous, is ANSYS. It's been around since the 1960's and it's the standard tool for large companies. This software can be used without barely interacting with the math underneath, one can simply design or import the domain, chose a package (for instance, Fluid Dynamics) and solve it. The user doesn't need to know about Lagrange Polynomials or quadrature methods.

In conclusion, one can find an uncountable number of tools to solve partial differential equations (PDE), and solving linear PDE is not cutting edge technology. Nevertheless, it is a worthy challenge for a programmer or engineer to turn the fairly abstract mathematical concepts into a working piece of code.

CHAPTER 2

The Boundary Value Problem

2.1 Partial Differential Equations

The purpose of the Finite Element Method is ultimately to solve partial differential equations. In this chapter we'll see the mathematical principles behind them, as well as a few example physical problems where they arise.

The Laplace Equation

A differential equation is any equation which relates a function to its derivatives. However the ones we are interested in are partial differential equations. In these, the function to be differentiated depends on multiple variables. Perhaps the simplest example is the Laplace equation:

$$\Delta u = 0 \tag{2.1}$$

where $u = u(x, y, z, t...)$, although for the sake of simplicity we'll stick to a two dimensional, stationary model, that is $u = u(x, y)$. The solutions to this equation are called the harmonic functions. They are very well known and therefore can be used as a test case for our FEM solver. This equation can be rewritten as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

The Poisson Equation

Taking it a step further we can make the Laplace equation non-homogeneous:

$$-\Delta u = f$$

where f is a function $f(x, y, z, t, ...)$. This function can be applied in many fields of physics such as electrostatics, thermal conduction, gravitation, etc.

A more general PDE

Because of thermal conductivity problems, the Poisson equation very often is made more complicated by adding a function in the first term:

$$-\nabla(a\nabla u) = f$$

If the heat problem deals with air-cooling, then another term is sometimes added:

$$-\nabla(a\nabla u) + c\,u = f$$

At this point, we have terms for u and its second derivative, so it makes sense to add a term for its first derivative in order to generalize it:

$$-\nabla(a\nabla u) + B\cdot\nabla u + c\,u = f$$

This is the equation we will solve in this project via the finite element method. Just an equation, however, is insufficient to fully define the problem. We must establish a domain, Ω with a boundary $\Gamma = \partial\Omega$ and certain boundary conditions. This is why this problem is also known as the Boundary Value Problem.

2.2 Boundary Conditions

A boundary condition is a condition on the solution of the PDE, forcing it or some of its derivatives to take a certain value. A very simple example would be that of a classical beam bending test. If we are studying small vertical deviation, we know a priori that, since both ends are on supports, they will not move. However it can get more complex. I'll present three types of boundary condition, as well as a hybrid between them.

A *Dirichlet boundary condition* is that which forces a certain value or function on the boundary. It is the simplest. For the Laplace Equation it looks like:

$$\begin{aligned} \Delta u &= 0 \quad \text{in } \Omega \\ u &= g \quad \text{on } \Gamma \end{aligned}$$

where g is a real-valued function at the border.

A *Neumann boundary condition*, on the other hand, imposes the value of the partial derivative of u perpendicular to the boundary. For the Laplace Equation it looks like:

$$\begin{aligned} \Delta u &= 0 \quad \text{in } \Omega \\ \frac{\partial u}{\partial n} &= h \quad \text{on } \Gamma \end{aligned} \tag{2.2}$$

where h is also a real-valued function at the border.

A *Robin boundary condition* is a boundary condition that imposes a relationship between the function and its derivative perpendicular to the boundary. For the Laplace Equation it looks like:

$$\begin{aligned} \Delta u &= 0 \quad \text{in } \Omega \\ c_0 u + c_1 \frac{\partial u}{\partial n} &= m \quad \text{on } \Gamma \end{aligned} \tag{2.3}$$

where c_0, c_1 and m are real-valued functions at the border.

A set of *mixed boundary conditions* is that which imposes a boundary condition on a section of the border Γ_1 and another type on another Γ_2. These two border sections must be complimentary and non-overlapping. An example for the Laplace Equation, with Dirichlet and Neumann BC would look like:

$$\begin{aligned} \Delta u &= 0 \quad \text{in } \Omega \\ u &= g \quad \text{on } \Gamma_1 \\ \frac{\partial u}{\partial n} &= h \quad \text{on } \Gamma_2 \end{aligned} \tag{2.4}$$

Although we've presented all of them for the Laplace equation, it must be clear that this concept applies for all PDE.

These are not the only boundary conditions that exist. For instance, the Cauchy boundary condition specifies both u and $\partial_n u$. When using it we run into the danger of over-specifying our problem. A clear example is the wall-attached beam. On the side it is attached to a wall, it neither moves (Dirichlet) nor rotates (Neumann). In contrast, we have no information of the opposite end. Since one end is over-defined and the other under-defined, the problem is overall properly formulated. These issues are beyond the scope of this project so our program will not consider Cauchy boundary conditions.

2.3 Physical Modelling

Heat flow

The PDE for heat flow can become complicated very easily, but for a steady state experiment it is the following:

$$\nabla \cdot (\kappa \nabla u) + q_{in} = 0 \tag{2.5}$$

Let's explain the terms here.

- u is the temperature and the function for which we want to solve.

κ is the heat conduction coefficient, a tabulated empirical positive value that depends on material and temperature, in other words, $\kappa = \kappa(x, y, z, u)$.

q_{in} is a function on Ω, and represents an internal heat source.

For homogeneous materials within a narrow range of temperatures, the equation can be simplified down to the following expression:

$$-\Delta u = \frac{q_{in}}{\kappa} \tag{2.6}$$

This is equivalent to Poisson's equation:

$$-\Delta u = f$$

I will now present the first problem you might face in a heat transfer course. Try to figure out the PDE and boundary conditions. For this you need to know the relationship between heat flow and temperature:

$$q_n = -\kappa \frac{\partial u}{\partial n} \tag{2.7}$$

There is a solid, homogeneous cylinder of length L and radius R, in a steady state. The bottom is held constant at 300K and the top at 350K, the side is adiabatic. What is its temperature distribution?

This case is even simpler, there is no internal heat source, hence $q_{in} = 0$. The boundary conditions are obvious for the top and bottom boundaries, they are Dirichlet BC. On all others, "adiabatic" means no heat flows across them, hence the normal derivative of u is zero. It is therefore a Neumann BC. Because we have two different BC, we know this is a mixed BC and the equation is:

$$\begin{aligned} \Delta u &= 0 && \text{in } \Omega \\ u &= 300 && \text{on } z = 0 \\ u &= 320 && \text{on } z = L \\ \frac{\partial u}{\partial n} &= 0 && \text{on } x^2 + y^2 = R^2 \end{aligned}$$

Due to axial symmetry, this is a one-dimensional problem, where the only axis that matters is the longitudinal axis:

$$\begin{aligned} \frac{\partial^2 u}{\partial z^2} &= 0 && \text{in } 0 < z < L \\ u &= 300 && \text{on } z = 0 \\ u &= 320 && \text{on } z = L \end{aligned} \tag{2.8}$$

Let's explore a two-dimensional problem:

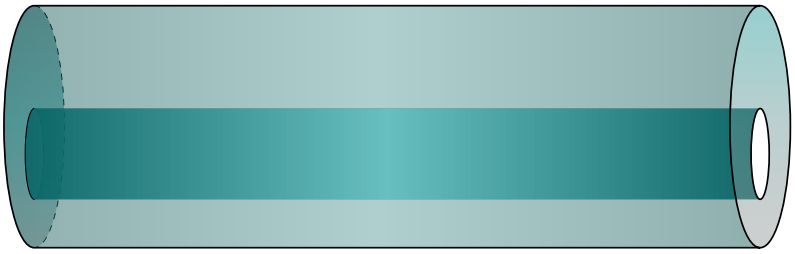

Figure 2.1: Heat transfer in an axially asymmetrical tube

There is an axially asymmetrical tube (figure 2.1) far longer than it is wide. The outside is in contact with cold air (T_a). Through it flows hot water (T_w). What is the temperature distribution within its cross-section?

Here the BC becomes a bit more complicated. Accurately calculating heat transfer onto a fluid requires modelling the flow of the fluid, however we can sacrifice some accuracy to simplify the calculation. The equation that approximates heat transfer between a solid and fluid is the following:

$$q_n = -\alpha(T_f - u)$$

Here, α is the convection coefficient, it depends on many factors and is usually tabulated. It relates the temperature difference between fluid and solid with the heat transferred between them. The other new variable is T_f, the temperature of the fluid. Developing it with the help of equation 2.7 we obtain:

$$\kappa \frac{\partial u}{\partial n} = \alpha(T_f - u)$$

Rearranging it becomes:

$$u + \frac{\kappa}{\alpha}\frac{\partial u}{\partial n} = T_f \tag{2.9}$$

We see that it is a Robin boundary condition. We now have all we need to solve the BVP, which becomes:

$$\begin{aligned} \Delta u &= 0 && \text{in } \Omega \\ u + \frac{\kappa}{\alpha_a}\frac{\partial u}{\partial n} &= T_a && \text{on } \Gamma_{\text{out}} \\ u + \frac{\kappa}{\alpha_w}\frac{\partial u}{\partial n} &= T_w && \text{on } \Gamma_{\text{in}} \end{aligned} \tag{2.10}$$

Where α_a and α_w are tabulated values for natural air convection and forced water convection, respectively.

Fluid dynamics

The Navier-Stokes Equations are the ten commandments of fluid dynamics, except fortunately there's only three of them. These equations will not be solved in this project, since the function u is a vector field and we'll only work with scalars. It is formative, however, to see that the previously explained concepts apply here as well.

To keep things simple, I will consider incompressible, Newtonian fluids. The first of the Navier-Stokes equation is the most elemental: mass conservation.

If we imagine a cubic volume differential in the middle of a fluid, we can see that the flow rate entering through the YZ face flowing parallel to the X axis equals

$$dQ_x = u_x dS_x = u_x dy\,dz \tag{2.11}$$

Where Q is the volumetric flow rate (i.e. the volume of fluid flowing per second), u_x is the flow velocity along the x-direction, and $dS_x = dy\,dz$. The flow that will exit on the opposite side will be:

$$dQ_{x+dx} = (u_x + \frac{\partial u_x}{\partial x} dx) dy\,dz$$

The combined outflow of the volume differential through both XY faces will be:

$$dQ_{x+dx} - dQ_x = \frac{\partial u_x}{\partial x} dx\,dy\,dz = \frac{\partial u_x}{\partial x} dV$$

If we consider all three axes and therefore all six faces of the volume differential, we obtain the total outflow of the volume differential. Because mass must be conserved, inputs and outputs must cancel out, so $Q = 0$. The expression then becomes:

$$\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} = 0 \tag{2.12}$$

This can be rewritten as:

$$\nabla \cdot \vec{u} = 0 \tag{2.13}$$

Here we see that incompressible fluids are non-divergent. The other equations are more complex so we will obtain them through a less rigorous approach and more by analogy. The second Navier-Stokes equation ensures conservation of momentum. It actually is a equation system of two (for 2D) or three (for 3D) equations:

$$u_x \frac{\partial u_i}{\partial x} + u_y \frac{\partial u_i}{\partial y} + u_z \frac{\partial u_i}{\partial z} = -\frac{\partial p}{\partial x_i} + \mu \Delta \vec{u} + \rho g_i \qquad i = x, y, z \tag{2.14}$$

This is often abbreviated to a single expression:

$$(\vec{u} \cdot \nabla)\vec{u} = -\nabla p + \mu \Delta \vec{u} + \rho \vec{g} \tag{2.15}$$

This equation is a convoluted way of saying $ma = \Sigma F$. The left side represents the acceleration, and the right side the different forces: pressure p, viscous forces (with viscous coefficient μ) and gravity g.

This is a very complicated set of equations to solve, however very often one can ditch some of the terms. For instance, in many applications gravity is negligible, and pressure gradients exist in only one direction.

Although there is a third Navier-Stokes equation, it is unnecessary for incompressible Newtonian flows, so we'll ignore it. We have four unknowns u_x, u_y, u_z and p; and four (1+3) equations:

$$\begin{aligned} \vec{\nabla} \cdot \vec{u} &= 0 \\ (\vec{u} \cdot \vec{\nabla})\vec{u} &= -\nabla p + \mu \Delta \vec{u} + \rho \vec{g} \end{aligned} \tag{2.16}$$

Let's explore now some possible boundary conditions. Dirichlet BC will appear in solid surfaces: a fluid cannot flow through a wall therefore

$$\vec{u} \cdot \vec{n} = 0 \qquad \text{on } \Gamma_{wall}$$

For non-idealized flows, there's also what's called the no-slip condition: the flow in contact with a wall moves at the same speed as the wall, i.e. the flow cannot slip past it, hence:

$$\text{proj}_\Gamma \vec{u} = \vec{v}_{wall} \qquad \text{on } \Gamma_{wall}$$

In the vast majority of the cases we deal with static walls, and therefore the previous two BC simplify to

$$\vec{u} = 0 \qquad \text{on } \Gamma_{wall}$$

We have simpler Dircichlet BC for pressure, which is often known at certain positions. For instance, the water-air interphase of an open channel problem is always at atmospheric pressure.

Neumann boundary conditions are rare in fluid mechanics, but can also exist. For example, if we know there is a plane or axis of symmetry we can impose that derivatives across it are 0.

Elastic Membrane

Consider a horizontal elastic net covering the unit square

$$Q = \{(x, y) \in \mathbb{R}^2 \quad \text{such that } 0 \le x \le 1 \text{ and } 0 \le y \le 1\}$$

formed by elastic strings tied together at nodes (x_i, y_j), where

$$(x_i, y_j) = (ih, jh) \qquad \text{with} \qquad i, j = 0, 1, 2, \ldots, n \quad \text{and} \quad h = \frac{1}{n}$$

forming a uniform quadrilateral mesh. Assume that the net is stretched so that the tension in each string is equal to h, corresponding to the tension being equal to one per unit of length. Note the normalization introduced says that the tension in each string decreases as the number of strings increases. We refer to the situation in which all the nodes lie in the plane of the square and there is no external load on the net as the unloaded reference configuration of the net.

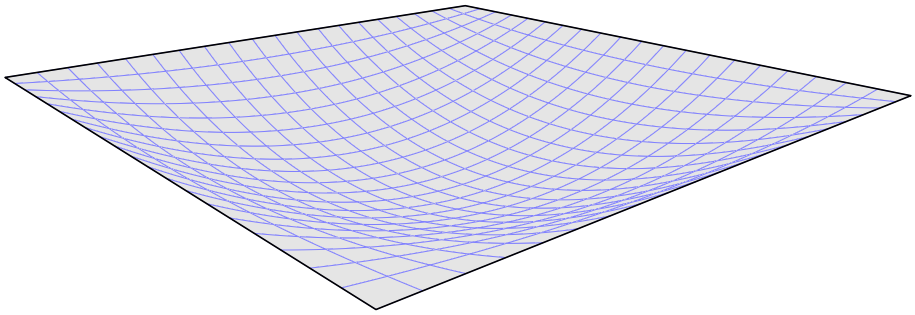

Figure 2.2: An elastic membrane under a load $f(x,y)$ in the square Q

Suppose the net is subject to a set of downward vertical loads of size $f_{i,j}h^2$ at the nodes (x_i, y_j) . The net will deform under the loads and the nodes will be displaced from the initial unloaded reference configuration. Let the vertical displacement of node (x_i, y_j) be denoted by $u_{i,j}$. If the displacements are small, then the vertical upward force from the net on node (x_i, y_j) is equal to

$$(u_{i,j}-u_{i-1,j})+(u_{i,j}-u_{i+1,j})+(u_{i,j}-u_{i,j-1})+(u_{i,j}-u_{,j+1})$$

with contributions from the four pieces of string meeting at (x_i, y_j). This is because the vertical slope of the line between for example nodes (x_i, y_j) and (x_i, y_{j+1}) is equal to

$$\frac{u_{i,j+1}-u_{i,j}}{h}$$

and the tension is h. We thus obtain the following vertical equilibrium equation for each node

$$-\frac{u_{i-1,j}-2u_{i,j}+u_{i+1,j}}{h^2}-\frac{u_{i,j-1}-2u_{i,j}+u_{i,j+1}}{h^2}=f_{i,j}$$

and passing to the limit when h tends to zero, by the Taylor formula, we obtain

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}=f \qquad \text{or} \qquad -\Delta u=f\,.$$

This equation expresses the equilibrium of a horizontal membrane made by an elastic fabric and carrying a vertical load of intensity (force per unit area) $f(x,y)$, where $u(x,y)$ is the vertical displacement of the membrane at (x,y) and we assume that

the membrane in its unloaded plane reference configuration is stressed to uniform tension in all directions.

We can generalize to a horizontal membrane covering a general domain Ω in $\mathbb{R}^2$. Assuming the membrane is fixed at the boundary Γ of Ω, so that the vertical displacement $u(x,y)$ is zero at Γ, we thus obtain the homogeneous Poisson's equation

$$\begin{cases} -\Delta u = f & \text{in} \quad \Omega \\ u = 0 & \text{on} \quad \Gamma. \end{cases}$$

CHAPTER 3

Mathematical tools

In this chapter we introduce the mathematical tools used in this project. As we will see in later chapters, we will solve Boundary Value Problems by the Finite Element Method. This requires going through the following steps:

- Obtain the variational formulation of the problem.
- Discretization of the problem using a space of functions of finite dimension.
- Assembly of the stiffness matrix and load vector.
- Solve the discrete system.

To discretize the problem, we will use Lagrange polynomials of different degrees defined on segments or triangles and to obtain the stiffness matrix and load vector we have to compute some integrals by the Gauss method and store the results in sparse matrices.

3.1 Sparse Matrices

The natural idea to take advantage of the zeros of a matrix and their location was initiated by engineers in various disciplines. In the simplest case involving banded matrices, special techniques are straightforward to develop. Electrical engineers dealing with electrical networks in the 1960s were the first to exploit sparsity to solve general sparse linear systems for matrices with irregular structure. The main issue, and the first addressed by sparse matrix technology, was to devise direct solution methods for linear systems. These had to be economical, both in terms of storage and computational effort. Sparse direct solvers can handle very large problems that cannot be tackled by the usual *dense* solvers.

Essentially, there are two broad types of sparse matrices: structured and unstructured. A structured matrix is one whose nonzero entries form a regular pattern, often along a small number of diagonals. Alternatively, the nonzero elements may lie in blocks (dense sub-matrices) of the same size, which form a regular pattern, typically along a small number of (block) diagonals. A matrix with irregularly located entries is said to be irregularly structured. The best example of a regularly structured matrix is a matrix that consists of only a few diagonals. Finite difference matrices on rectangular grids are typical examples of matrices with regular structure. Most finite element or finite volume techniques applied to complex geometries lead to irregularly structured matrices.

In order to take advantage of the large number of zero elements, special schemes are required to store sparse matrices. The main goal is to represent only the nonzero elements, and to be able to perform the common matrix operations. In the following, n_z denotes the total number of nonzero elements, m the number of rows and n the number of columns of a matrix A.

The simplest storage scheme for sparse matrices is the so-called coordinate format or triplet form. The data structure consists of three arrays:

- an array containing all the values of the nonzero elements of A in any order;
- an integer array containing their row indices;
- a second integer array containing their column indices.

All three arrays are of length n_z, the number of nonzero elements.

Example 3.1 The matrix

$$A = \begin{pmatrix} 2 & 0 & 0 & -1 & 0 \\ 3 & 1 & 0 & 0 & 4 \\ 0 & 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & 2 & 0 \end{pmatrix},$$

can be represented by

Array	Values									
Data	2	−3	1	1	3	4	−1	−1	1	2
Row index	0	2	3	4	1	1	0	3	1	4
Column index	0	2	4	0	0	4	3	3	1	3

Table 3.1: Triplet form of a matrix

where the row indices are from 0 to $m-1$, and the column indices from 0 to $n-1$. This form is easily saved in a text file, where each line of the file contains one entry of the matrix (we can add a first line with the number of rows and columns of the matrix,

```
5 5
0 0 2
2 2 -3
3 4 1
4 0 1
1 0 3
1 4 4
0 3 -1
3 3 -1
1 1 1
4 3 2
```

In the above example, the elements are listed in an arbitrary order. In fact, they are usually listed by rows or by columns. If the elements were listed by column, the array "column index" which contains redundant information might be replaced by an array which points to the beginning of each column instead. This would involve non-negligible savings in storage. The new data structure has three arrays with the following properties:

- an array containing all the values of the nonzero elements of A listed column by column, from column 0 to column n. It has length n_z;
- an integer array containing their row indices;
- an integer array that contains the indices to the beginning of each column in the previous arrays. Thus, the content of this array is the position in the array of data where the i-th column starts. The length of this array is $m+1$, with the last element equal to n_z.

This format is probably the most popular for storing general sparse matrices. It is called the *Compressed Sparse Column* (CSC) format. This scheme is preferred over the coordinate scheme because it is often more useful for performing typical computations. On the other hand, the triplet form is advantageous for its simplicity and its flexibility. It is often used as an *entry* format in sparse matrix software packages.

The CSC representation of the above example can be seen in table 3.2.

Observe that the values in blue are the first element of each column, that the third array points to these elements and that the i-th entry of this array represents the number of non null elements on the first i rows of the matrix A. In this way, the CSC format can be extended to matrices with null columns. In the source code listing 3.1, we can see the structures and functions used for sparse matrices.

The *Compressed Sparse Row* (CSC) format is similar to CSC except that values are read first by row, a column index is stored for each value, and row pointers are stored.

Array	Values									
Data	2	3	1	1	−3	−1	−1	2	4	1
Row index	0	1	4	1	2	0	3	4	1	3
Column's first element index	0	3	4	5	8	10				

Table 3.2: Compressed Sparse Column form

```
typedef struct{
   int nz, m, n; //Number of entries, rows and columns

   //CSC
   int *Ap;      //Column-accumulated counter
   int *Ai;      //Row index
   double *Ax;   //Value stored

   //Triplet form
   int *Ti;         //i-value
   int *Tj;         //j-value
   double *Tx;      //(i,j) value
   size_t tmUsed;   //used memory for triplet form
   size_t tmAlloc; //allocated memory for triplet form
}Sparse;

//CSC functions
Sparse sparse_pack(int m, int n, double A[m][n]);
double ** sparse_unpack(Sparse *sparse);
void sparse_free(Sparse *A);
void sparse_print(Sparse *A);

//Triplet form functions
int triplet_append(Sparse *t, int i, int j, double x);
void triplet_initialize(Sparse *t, int size);
void triplet_free(Sparse *t);
void triplet_print(Sparse *t);
int triplet_store(Sparse *t);
int colFromTriplet(Sparse *A,const char *file, int line);
double **triplet_unpack(Sparse *K);

//Linear system solving functions
double *umfpack_solve(Sparse *A, double *b, const char *file, int line, int isTriplet);
double *GSL_solve(Sparse *K, double *F, size_t n, size_t max_iter, double tol);
```

Source code 3.1: Triplet and compressed sparse column forms

3.2 Piecewise polynomial interpolation. Dimension 1

Polynomials are used as the basic means of approximation in nearly all areas of numerical analysis. They are used in the solution of equations and in the approximation of functions, of integrals and derivatives, of solutions of integral and differential equations, etc. Polynomials owe this popularity to their simple structure, which makes it easy to construct effective approximations and then make use of them.

For this reason, the representation and evaluation of polynomials is a basic topic in numerical analysis. We discuss this topic in the present chapter in the context of

polynomial interpolation, the simplest and certainly the most widely used technique for obtaining polynomial approximations.

We can describe the basic problem of interpolation as follows: given a continuous real-valued function $f(x)$ defined in the interval $[a,b]$ and $n+1$ points

$$a \le x_0 < x_1 < x_2 < \cdots < x_n \le b\,,$$

we wish to construct a polynomial $p(x)$ of degree at most n such that

$$p(x_i) = f(x_i) \qquad \text{for} \quad i = 0,1,2,\ldots,n\,.$$

It's easy to show that this problem has a unique solution and that this solution can be expressed in terms of the Lagrange polynomials. Let

$$L_n^i(x) = \prod_{\substack{k=0\\k\neq i}}^{n} \frac{x-x_k}{x_i-x_k}\,,$$

then

$$p(x) = \sum_{k=0}^{n} f(x_i) L_n^i(x).$$

The polynomials $L_n^i(x)$ for $i = 0,1,\ldots,n$ are called the *basis Lagrange polynomials* for the points $x_0, x_1, \ldots, x_n$ and they verify the property

$$L_n^i(x_k) = \delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise.} \end{cases}$$

In figure 3.1, we can see the interpolating polynomial in the interval $[-4,4]$ of the function

$$f(x) = \frac{4}{1+x^2}$$

using nine points. This approximation yields an oscillating curve above and below the true function. This behavior tends to grow with the number of points, leading to a divergence known as Runge's phenomenon. The problem may be eliminated by choosing interpolation points at Chebyshev nodes.

A quite natural and different approach to approximate a function on an interval $[a,b]$ is to first split the interval into subintervals and then approximate the function by a polynomial of fairly low degree on each subinterval. To simplify the notation, we will split the initial interval into m non-overlapping subintervals of equal length $I_0, I_1, I_2, \ldots, I_{m-1}$, where

$$I_k = \big[a + kh, a + (k+1)h\big] \qquad \text{with} \qquad h = \frac{b-a}{m}\,,$$

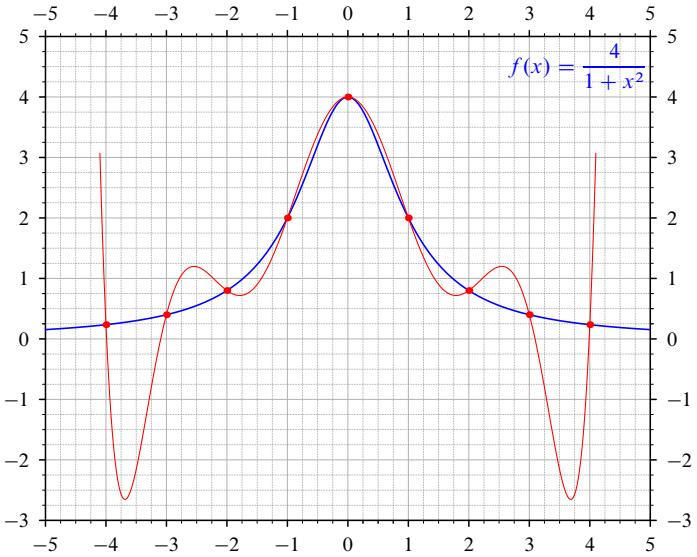

Figure 3.1: Interpolating polynomial of a function $f(x)$ with nine points

and then use interpolation with $n+1$ equally spaced points or nodes on each subinterval I_k, so that

$$I_k = [x_0^k, x_n^k] \qquad \text{with} \qquad x_0^k < x_1^k < x_2^k < \cdots < x_n^k$$

and

$$x_{i+1}^k - x_i^k = \frac{x_n^k - x_0^k}{n}.$$

For example, if we divide the interval $[a,b]$ into 6 subintervals and use interpolation polynomials of degree 4 on each subinterval, we will have 25 nodes.

We have to find the basis Lagrange polynomials for $n+1$ points on every subinterval I_k. In order to do that, we will start with the sample interval $I = [0,1]$ with the interpolation points

$$x_k = \frac{k}{n} \qquad k = 0, 1, \ldots, n\,.$$

It's easy to see that

$$\mathscr{L}_n^i(x) = \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{x - x_k}{x_i - x_k} = \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{x - \frac{k}{n}}{\frac{i}{n} - \frac{k}{n}} = \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{nx - k}{i - k} \tag{3.1}$$

In the figure 3.2, we can see the 5 Lagrange polynomials corresponding to the interval $[0,1]$ with 5 equally spaced node points.

In the general case, the interval is $I = [a,b]$ and the points

$$x_k = a + \frac{k(b-a)}{n} \qquad k = 0, 1, 2, \ldots, n\,,$$

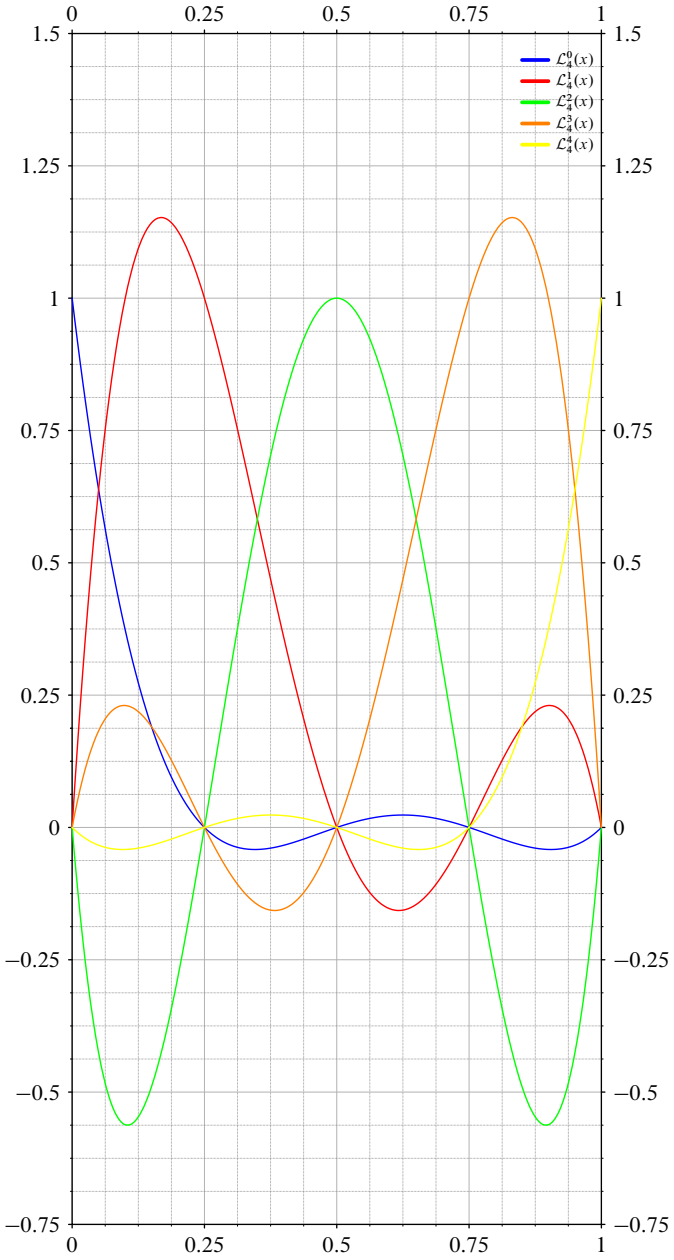

Figure 3.2: Lagrange polynomials $\mathscr{L}_4^i(x)$ for $i = 0, 1, 2, 3, 4$

then

$$x_i - x_k = a + \frac{i(b-a)}{n} - a - \frac{k(b-a)}{n} = \frac{(i-k)(b-a)}{n}$$

$$x - x_k = x - a - \frac{k(b-a)}{n} = \frac{nx - na - k(b-a)}{n}.$$

Finally, we have that

$$L_n^i(x) = \prod_{\substack{k=0 \\ i\neq k}}^{n} \frac{\frac{nx-na-k(b-a)}{n}}{\frac{(i-k)(b-a)}{n}} = \prod_{\substack{k=0 \\ i\neq k}}^{n} \frac{nx-na-k(b-a)}{(i-k)(b-a)}$$
$$= \prod_{\substack{k=0 \\ i\neq k}}^{n} \frac{\frac{n(x-a)}{b-a}-k}{i-k} = \mathscr{L}_n^i\left(\frac{x-a}{b-a}\right)$$

that is, the basis Lagrange polynomials for $n+1$ equally spaced points can be obtained by a change of variable from the basis Lagrange polynomial at the sample interval $[0,1]$.

The derivatives of the basis Lagrange polynomials are

$$\frac{d}{dx}\mathscr{L}_n^i(x) = n\sum_{\substack{k=0 \\ k\neq i}}^{n}\left(\frac{1}{i-k}\prod_{\substack{j=0 \\ j\neq i,k}}^{n}\frac{nx-k}{i-k}\right)$$

and

$$\frac{d}{dx}L_n^i(x) = \frac{1}{b-a}\frac{d}{dx}\mathscr{L}_n^i\left(\frac{x-a}{b-a}\right)$$

In the Finite Element Method in dimension one, given an interval $[a,b]$ and two integers m and n, we will divide the interval into m non-overlapping subintervals of equal length $I_0, I_1, I_2, \ldots, I_{m-1}$ and consider the vector space $\mathbb{P}_m^n([a,b])$ of all continuous functions on $[a,b]$ such that their restriction to I_k is a polynomial of degree at most n. It's not difficult to show that $\mathbb{P}_m^n([a,b])$ has dimension $m+1+m(n-1) = mn+1$ and that a function is determined by its values at the nodes

$$x_i = a+ih \qquad \text{for} \qquad i = 0,1,2,\ldots,mn \qquad \text{where} \qquad h = \frac{b-a}{mn}.$$

Moreover, the subintervals $I_0, I_1, I_2, \ldots, I_{m-1}$ are determined by

$$I_k = [x_{kn}, x_{(k+1)n}].$$

We will represent with $\phi_i(x)$ the function in $\mathbb{P}_m^n([a,b])$ such that his values at the nodes $x_0, x_1, x_2, \ldots, x_{mn}$ are

$$\phi_i(x_k) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise.} \end{cases}$$

These functions form a basis of the space $\mathbb{P}_m^n([a,b])$ that we'll call the *standard basis*.

For example, if we start with the interval $[1,7]$ and we choose $m=3$ and $n=1$, we have that $\mathbb{P}_3^1([1,7])$ has dimension 4, and we can see all four nodes $x_0=1$, $x_1=3$, $x_2=5$ and $x_3=7$ in figure 3.3.

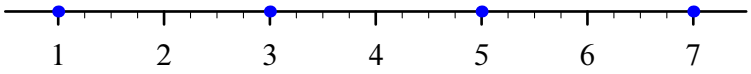

Figure 3.3: Nodes of the space $\mathbb{P}_3^1([1,7])$

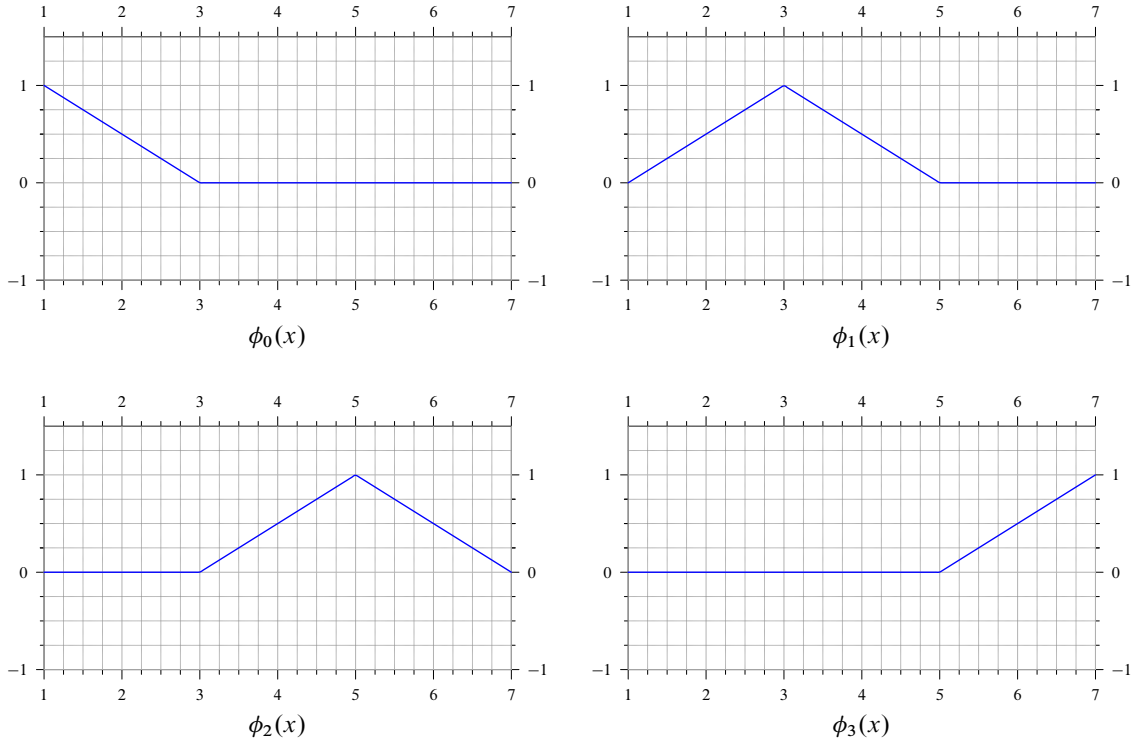

Figure 3.4: Basis functions of $\mathbb{P}_3^1([1,7])$

In figure 3.4 we can see the graphics of the standard basis of $\mathbb{P}_3^1([1,7])$.

In a similar way, if we consider the interval $[-1,5]$ and we choose $m=6$ and $n=4$, we have that $\mathbb{P}_6^4([-1,5])$ has dimension 25 and we can see the 25 nodes in the figure 3.5.

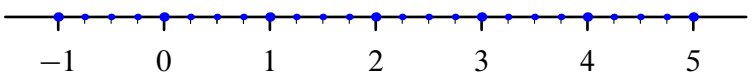

Figure 3.5: Nodes of the space $\mathbb{P}_6^4([-1,5])$

In the figure 3.6 we can see the graphics of part of the standard basis in $\mathbb{P}_6^4([-1,5])$. An important property of the basis functions $\phi_0, \phi_1, \phi_2, \ldots, \phi_{mn}$ is that they are non-zero in at most two subintervals I_k.

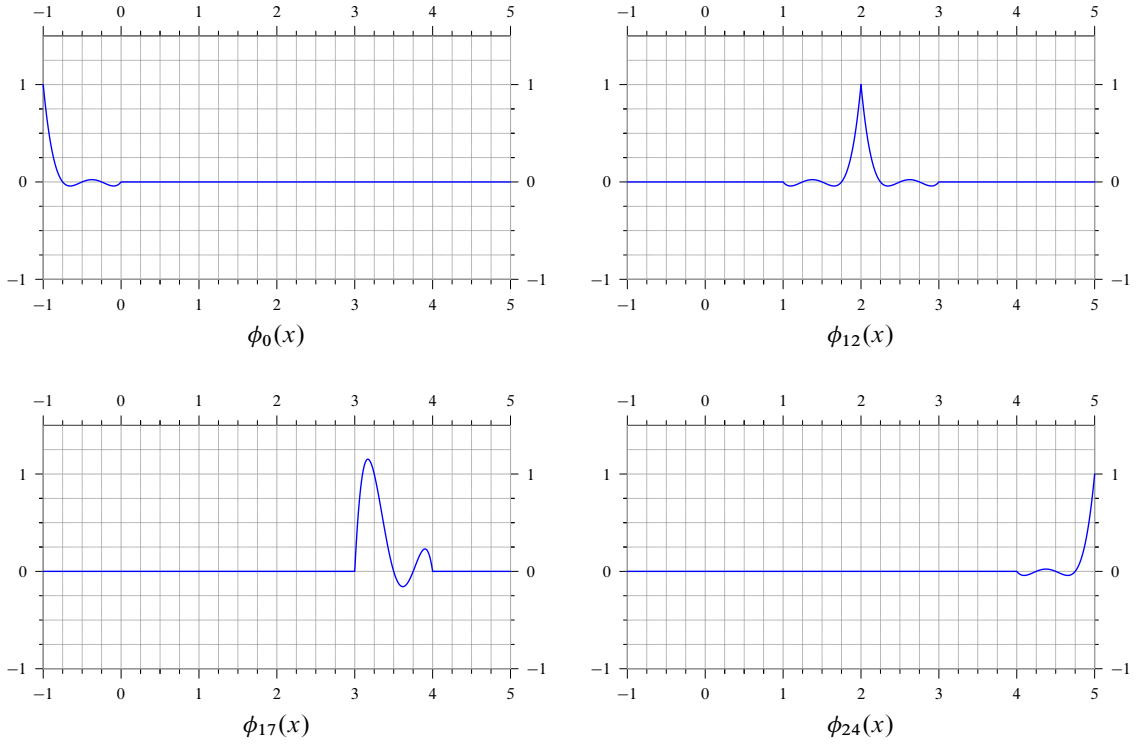

Figure 3.6: Some basis functions of $\mathbb{P}_6^4([-1,5])$

3.3 Piecewise polynomial interpolation. Dimension 2

In this chapter we extend the concept of piecewise polynomial approximation to the two dimensional plane. As before, the basic idea is to construct spaces of piecewise polynomial functions that are easy to represent in a computer and to show that they can be used to approximate more general functions. A difficulty with the construction of piecewise polynomials in higher dimension is that first the underlying domain must be partitioned into elements, such as triangles, which may be a nontrivial task if the domain has complex shape. This partition is explored in the next chapter. Here we present a methodology for building representations of piecewise polynomials on triangulations that is efficient and suitable for computer implementation and to study the approximation properties of these spaces.

In the one dimensional case, we have divided the interval $[a,b]$ in m subintervals, but in dimension 2, we will have much more complex domains. As we will see in later chapters, the first step is is to obtain a triangulation of the domain Ω. Figure 3.7 represents a triangulation of a domain with the shape of the letter A.

Then, given a triangulation $\mathscr{T}$ of the domain Ω, we consider the space $\mathbb{P}^n_{\mathscr{T}}(\Omega)$ of all continuous functions on Ω such that its restriction to any triangle $T \in \mathscr{T}$ is a polynomial of degree at most n.

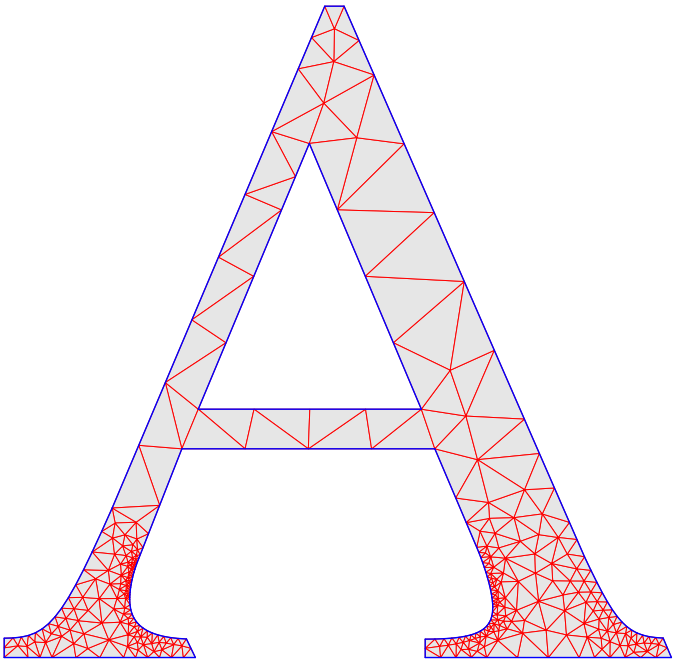

Figure 3.7: Triangulation of a domain

Since the basic triangle $T_{\mathscr{B}}$ with vertices $(0,0)$, $(1,0)$ and $(0,1)$ can be mapped into any triangle T with vertices (x_0, y_0), (x_1, y_1), (x_2, y_2) (see figure 3.8) with the transformation $h(x,y)$ given by

$$\begin{pmatrix} t \\ s \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \tag{3.2}$$

we will define a basis for the space $\mathbb{P}^n(T_{\mathscr{B}})$ of the polynomials of degree at most n in the basic triangle $T_{\mathscr{B}}$. Any polynomial of degree at most n with two variables can be written as

$$p(x) = \sum_{i=0}^{n} \sum_{j=0}^{n-i} a_{i,j}\, x^i\, y^j$$

and it has $1+2+3+\cdots+n+1$ coefficients, then the dimension of $\mathbb{P}^n(T_{\mathscr{B}})$ is

$$d_n = \dim \mathbb{P}^n(T_{\mathscr{B}}) = \frac{(n+1)(n+2)}{2}.$$

Moreover, every polynomial $p(x,y)$ of degree at most n is uniquely determined by its values at suitable d_n points or nodes. We will use the following points

$$x_{i,j} = \left(\frac{i}{n}, \frac{j}{n}\right) \qquad \text{for} \qquad i = 0,1,\ldots,n \qquad j = 0,1,\ldots,n-i$$

and for every point pair (i,j), let $\mathscr{L}_n^{i,j}(x,y)$ be the polynomial such that

$$\mathscr{L}_n^{i,j}(x_{k,m}) = \begin{cases} 1 & \text{if } k = i \text{ and } m = j \\ 0 & \text{otherwise.} \end{cases}$$

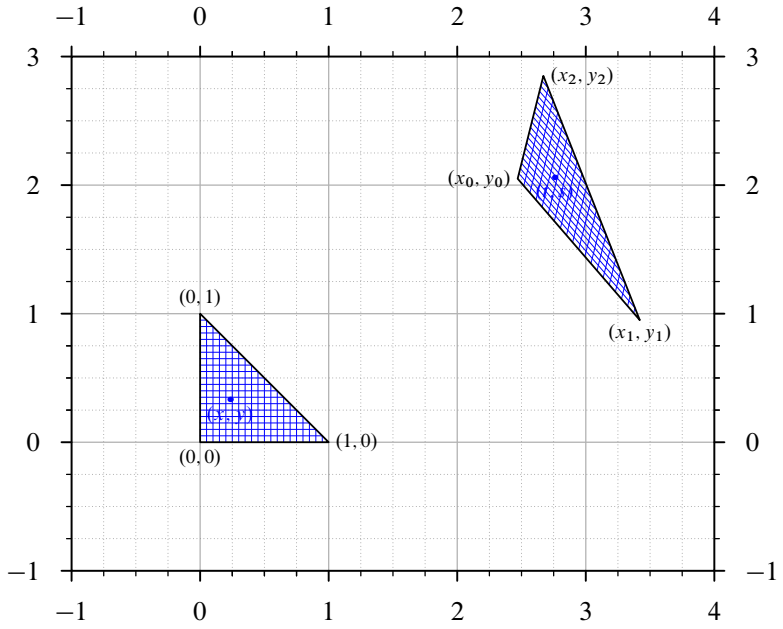

Figure 3.8: Map from triangle $T_{\mathscr{B}}$ to triangle T

For $n=5$, we have 21 nodes, as we can see in figure 3.9, and it's easy to check that the polynomial

$$p(x)=x\,y\Big(y-\frac{1}{5}\Big)(x+y-1)\Big(x+y-\frac{4}{5}\Big)$$

has degree 5, it vanishes at all nodes except $x_{1,2}$ (the red color point), where it evaluates to a non-zero value.

In general, given a node $x_{i,j}$, we can define

$$L_n^{i,j}(x,y)=\prod_{p=0}^{i-1}\Big(x-\frac{p}{n}\Big)\prod_{p=0}^{j-1}\Big(y-\frac{p}{n}\Big)\prod_{p=i+j+1}^{n}\Big(x+y-\frac{p}{n}\Big)$$

which vanishes at all nodes of $T_{\mathscr{B}}$ except at node $x_{i,j}$. Then, we can obtain that

$$\begin{aligned}\mathscr{L}_n^{i,j}(x,y)&=\frac{L_n^{i,j}(x,y)}{L_n^{i,j}\left(\frac{i}{n},\frac{j}{n}\right)}\\&=\prod_{k=0}^{i-1}\left(\frac{n\,x-k}{i-k}\right)\prod_{k=0}^{j-1}\left(\frac{n\,y-k}{j-k}\right)\prod_{k=i+j+1}^{n}\left(\frac{n\,x+n\,y-k}{i+j-k}\right)\end{aligned}\tag{3.3}$$

The $\mathscr{L}_n^{i,j}(x,y)$ for $i=0,1,2,\ldots,n$ and $j=0,1,2,\ldots,i-1$ are the *basis Lagrange polynomials* for $T_{\mathscr{B}}$.

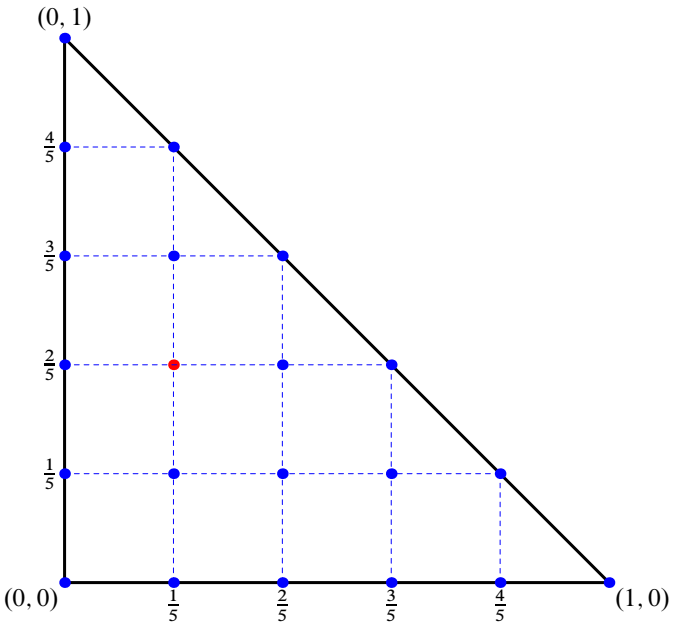

Figure 3.9: Nodes on $T_{\mathscr{B}}$ for $n=5$

To compute their partial derivatives, we write

$$F(x)=\prod_{k=0}^{i-1}\left(\frac{nx-k}{i-k}\right)$$
$$G(y)=\prod_{k=0}^{j-1}\left(\frac{ny-k}{j-k}\right)$$
$$H(x,y)=\prod_{k=i+j+1}^{n}\left(\frac{nx+ny-k}{i+j-k}\right),$$

then, $\mathscr{L}_n^{i,j}(x,y)=F(x)G(y)H(x,y)$ and, by the rules of the derivative, we get that

$$\begin{aligned}\frac{\partial}{\partial x}\mathscr{L}_n^{i,j}(x,y)&=F'(x)G(y)H(x,y)+F(x)G(y)\frac{\partial}{\partial x}H(x,y)\\&=nG(y)H(x,y)\sum_{k=0}^{i-1}\left(\frac{1}{i-k}\prod_{\substack{p=0\\p\neq k}}^{i-1}\frac{nx-p}{i-p}\right)\\&+nF(x)G(y)\sum_{k=i+j+1}^{n}\left(\frac{1}{i+j-k}\prod_{\substack{p=i+j+1\\p\neq k}}^{n}\frac{nx+ny-p}{i-p}\right)\end{aligned}\tag{3.4}$$

and

$$\begin{aligned}\frac{\partial}{\partial y}\mathscr{L}_n^{i,j}(x,y) &= F(x)G'(y)H(x,y)+F(x)G(y)\frac{\partial}{\partial y}H(x,y)\\ &= nF(x)H(x,y)\sum_{k=0}^{j-1}\left(\frac{1}{j-k}\prod_{\substack{p=0\\p\neq k}}^{j-1}\frac{ny-p}{j-p}\right)\\ &+ nF(x)G(y)\sum_{k=i+j+1}^{n}\left(\frac{1}{i+j-k}\prod_{\substack{p=i+j+1\\p\neq k}}^{n}\frac{nx+ny-p}{i+j-p}\right)\end{aligned} \tag{3.5}$$

Let's consider again triangle T with vertices (x_0,y_0), (x_1,y_1), (x_2,y_2) and the vectors $\vec{v}_1=(x_1-x_0,y_1-y_0)$ and $\vec{v}_2=(x_2-x_0,y_2-y_0)$. In triangle T, we define the points

$$p_{i,j}=(x_0,y_0)+i\,\vec{v}_1+j\,\vec{v}_2 \qquad \text{for} \qquad i=0,1,\ldots,n \qquad j=0,1,\ldots,n-i\,,$$

and for every point pair (i,j), let $L_n^{i,j}(x,y)$ be the polynomial such that

$$L_n^{i,j}(p_{k,m})=\begin{cases}1 & \text{if } k=i \text{ and } m=j\\ 0 & \text{otherwise.}\end{cases}$$

These polynomials are a basis for $\mathbb{P}^n(T)$ and is easy to see that $L_n^{i,j}(t,s)=\mathscr{L}_n^{i,j}(h^{-1}(t,s))$ where h is the transformation defined in 3.2. As we will see in later chapters, in the finite element method we need to compute the partial derivatives of the functions $L_n^{i,j}(t,s)$ and we have chosen to obtain them from the derivatives of $\mathscr{L}_n^{i,j}(x,y)$.

Due to the chain rule, we have that (note that we write only L and $\mathscr{L}$ for simplicity)

$$\frac{\partial}{\partial t}L(t,s)=\frac{\partial}{\partial x}\mathscr{L}\big(h^{-1}(t,s)\big)\frac{\partial}{\partial t}x(t,s)+\frac{\partial}{\partial y}\mathscr{L}\big(h^{-1}(t,s)\big)\frac{\partial}{\partial t}y(t,s)$$

and

$$\frac{\partial}{\partial s}L(t,s)=\frac{\partial}{\partial x}\mathscr{L}\big(h^{-1}(t,s)\big)\frac{\partial}{\partial s}x(t,s)+\frac{\partial}{\partial y}\mathscr{L}\big(h^{-1}(t,s)\big)\frac{\partial}{\partial s}y(t,s).$$

From 3.2, we have that

$$\begin{aligned}\begin{pmatrix}x\\y\end{pmatrix}&=\begin{pmatrix}x_1-x_0 & x_2-x_0\\ y_1-y_0 & y_2-y_0\end{pmatrix}^{-1}=\begin{pmatrix}a & b\\ c & d\end{pmatrix}^{-1}\begin{pmatrix}t-x_0\\ s-y_0\end{pmatrix}\\ &=\frac{1}{D}\begin{pmatrix}d & -c\\ -b & a\end{pmatrix}\begin{pmatrix}t-x_0\\ s-y_0\end{pmatrix},\end{aligned}$$

then

$$\begin{aligned}\frac{\partial}{\partial t}L(t,s)&=\frac{1}{D}\Big(d\,\frac{\partial}{\partial x}\mathscr{L}\big(h^{-1}(t,s)\big)-c\,\frac{\partial}{\partial y}\mathscr{L}\big(h^{-1}(t,s)\big)\Big)\\ \frac{\partial}{\partial s}L(t,s)&=\frac{1}{D}\Big(-b\,\frac{\partial}{\partial x}\mathscr{L}\big(h^{-1}(t,s)\big)+a\,\frac{\partial}{\partial y}\mathscr{L}\big(h^{-1}(t,s)\big)\Big).\end{aligned}$$

To compute the values of the Lagrange polynomials at the different intervals or triangles, we store their values in the Gauss quadrature points for the basic interval $[0,1]$ or the basic triangle $T_{\mathscr{B}}$. We can see the functions used in the listing 3.2.

```
1  // One dimension:
2  typedef struct{
3     double value;
4     double dvalue;
5  } lagrange_1D_value;
6
7  typedef lagrange_1D_value **Lagrange;
8
9  Lagrange lagrangeAtGaussPoints(int deg, int npoints, gauss_qdat *qdat);
10
11 //Two dimensions
12 typedef struct{
13    double value;
14    double dxValue;
15    double dyValue;
16 } lagrange_2D_value;
17
18 typedef lagrange_2D_value ***Lagrange2D;
19
20 Lagrange2D lagrangeAtTWBPoints(int deg, int npoints, TWB_qdat *qdat);
```

Source code 3.2: Functions used to work with Lagrange polynomials

3.4 Gauss Quadrature

In this section we are going to explore various ways for approximating the integral of a function over a given domain. Since we can not analytically integrate every function, the need to approximate integration formulas is self-evident. In addition, there might be situations where the given function can be integrated analytically, and still, an approximation formula may end up being a more efficient alternative to evaluating the exact expression of the integral.

The first option to compute an integral

$$\int_a^b f(x)\,dx$$

is based on the interpolation polynomials. The basic idea is to select a set of distinct nodes $x_0, x_1, x_2, \ldots, x_n$ from the interval $[a,b]$ and then integrate the Lagrange interpolating polynomial,

$$\begin{aligned}\int_a^b f(x)\,dx \simeq \int_a^b P_n(x)\,dx &= \int_a^b \sum_{i=0}^{n} f(x_i)L_i(x)\,dx \\ &= \sum_{i=0}^{n} f(x_i)\int_a^b \prod_{\substack{k=0\\k\neq i}}^{n} \frac{x-x_k}{x_i-x_k} = \sum_{i=0}^{n} A_i f(x_i),\end{aligned}$$

where

$$A_i = \int_a^b \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{x - x_k}{x_i - x_k}.$$

For equally spaced points, $x_0, x_1, x_2, \ldots, x_n$, a numerical integration formula of the form

$$\int_a^b f(x)\,dx \simeq \sum_{i=0}^{n} A_i f(x_i)$$

is called a *Newton-Cotes* formula.

We can also use a piecewise polynomial interpolation, that is divide the interval $[a, b]$ into m subintervals $I_0, I_1, I_2, \ldots, I_{m-1}$ and then use a polynomial interpolation of degree n for each subinterval. The most common cases are $n = 1$ or trapezoidal rule

$$\int_a^b f(x)\,dx \simeq \frac{h}{2}\Big(fa) + \sum_{i=1}^{m-1} f(a+ih) + f(b)\Big) \qquad \text{where} \qquad h = \frac{b-a}{m}$$

and $n = 2$, or Simpson rule,

$$\int_a^b f(x)\,dx \simeq \frac{h}{3}\Big(f(a) + \sum_{i=1}^{\frac{m}{2}-1} 2f(a+2ih) + \sum_{i=1}^{\frac{m}{2}} 4f\big(a+(2i+1)h\big) + f(b)\Big)$$

where

$$h = \frac{b-a}{2m}$$

It can be seen that any quadrature formula obtained from interpolating polynomials of degree n is exact for all polynomials of degree at most n.

A second option is to use *Gaussian quadratures*. Here, we try to find a quadrature formula

$$\int_a^b f(x)\,dx \simeq \sum_{i=0}^{n} A_i f(x_i) \tag{3.6}$$

choosing the points $x_0, x_1, x_2, \ldots, x_n$ in an optimal way, rather that equally spaced. To measure this accuracy, we assume that the best choice of the points and coefficients A_i produces the exact result for the largest class of polynomials. Then, we have $2n$ parameters to choose, and we can reasonable expect that they can be obtained so that the formula 3.6 will be exact for all polynomials of degree at most $2n-1$.

In order to describe the Gaussian quadrature formula, we need the Legendre polynomials. They are an infinite collection of polynomials $P_0(x), P_1(x), \ldots, P_n(x), \ldots$ such that

(a) $P_n(x)$ has degree n and $P_n(1) = 1$.

(b) They are orthogonal, that is,

$$\int_{-1}^{1} P_m(x)P_n(x)\,dx = 0 \qquad \text{if} \qquad m \neq n\,.$$

For example, the first six Legendre polynomials are

$$\begin{aligned} P_0(x) &= 1 & P_1(x) &= x \\ P_2(x) &= \frac{3x^2-1}{2} & P_3(x) &= \frac{5x^3-3x}{2} \\ P_4(x) &= \frac{35x^4-30x^2+3}{8} & P_5(x) &= \frac{63x^5-70x^3+15x}{8}\,. \end{aligned}$$

The roots of these polynomials are distinct, lie in the interval $(-1,1)$, have a symmetry with respect to the origin, verify the recurrent relation

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

and, most importantly, are the correct choice for determining the parameters that give us the nodes and coefficients for our quadrature method.

Theorem 3.2 *The Gaussian quadrature rule for the interval* $[-1,1]$

$$\int_{-1}^{1} f(x)\,dx \simeq \sum_{i=1}^{n} A_i f(\zeta_i)$$

where $\zeta_1, \zeta_2 \ldots, \zeta_n$ *are the roots of the Legendre polynomial* P_n *and*

$$A_i = \frac{2}{(1-\zeta_i^2)P_n'(\zeta_i)^2}$$

is exact for all polynomials of degree $2n-1$.

If we want to evaluate the integral of a function $f(x)$ over an interval $[a,b]$, we can use a variable substitution:

$$x = \frac{(b-a)t}{2} + \frac{b+a}{2}$$

and we get

$$\begin{aligned} \int_a^b f(x)\,dx &= \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{(b-a)t}{2} + \frac{b+a}{2}\right)dt \\ &\simeq \frac{b-a}{2}\sum_{i=1}^{n} A_i f\left(\frac{(b-a)\zeta_i}{2} + \frac{b+a}{2}\right). \end{aligned}$$

n	Roots ζ_i	Values of A_i
$n=2$	−0.5773502691896	1.0
	0.5773502691896	1.0
$n=3$	−0.7745966692415	0.5555555555555
	0.0	0.8888888888889
	0.7745966692415	0.5555555555555
$n=4$	−0.8611363115941	0.3478548451375
	−0.3399810435849	0.6521451548625
	0.3399810435849	0.6521451548625
	0.8611363115941	0.3478548451375
$n=5$	−0.9061798459387	0.2369268850562
	−0.5384693101057	0.4786286704994
	0.0	0.5688888888889
	0.5384693101057	0.4786286704994
	0.9061798459387	0.2369268850562

Table 3.3: Gauss quadrature data from [8]

The constants A_i needed for the quadrature rule can be generated from the equation in Theorem 3.2, but both these constants and the roots of the Legendre polynomials are extensively tabulated. Table 3.3 lists these values for $n = 2, 3, 4$ and 5.

In the application of the Finite Element Method, we need to compute integrals of the form

$$\int_a^b f(x) L_n^i(x)\, dx$$

where $L_i^n(x)$ is the i-th Lagrange polynomial of degree n at the interval $[a, b]$ with equally spaced points

$$x_k = a + \frac{k(b-a)}{n} \qquad \text{for} \qquad k = 0, 1, 2, \ldots, n\,.$$

If we transform the interval $[a, b]$ to $[0, 1]$ with the change of variable

$$x = (b-a)t + a$$

as a first step, and then transform the interval $[0, 1]$ to $[1, 1]$ by substituting

$$t = \frac{s+1}{2}$$

as a second step, we get that

$$\begin{aligned}\int_a^b f(x)L_n^i(x)\,dx &= (b-a)\int_0^1 f\big((b-a)t+a\big)L_n^i\big((b-a)t+a\big)\,dt\\ &= (b-a)\int_0^1 f\big((b-a)t+a\big)\mathscr{L}_n^i(t)\,dt\\ &= \frac{b-a}{2}\int_{-1}^1 f\Big(\frac{(b-a)s}{2}+\frac{a+b}{2}\Big)\mathscr{L}_n^i\Big(\frac{s+1}{2}\Big)\,ds\\ &\simeq \frac{b-a}{2}\sum_{i=1}^m A_i f\Big(\frac{(b-a)\zeta_i}{2}+\frac{a+b}{2}\Big)\mathscr{L}_n^i\Big(\frac{\zeta_i+1}{2}\Big).\end{aligned} \tag{3.7}$$

In a similar way, we get that

$$\begin{aligned}\int_a^b f(x)\frac{d}{dx}L_n^i(x)\,dx &= (b-a)\int_0^1 f\big((b-a)t+a\big)\frac{d}{dx}L_n^i\big((b-a)t+a\big)\,dt\\ &= (b-a)\int_0^1 f\big((b-a)t+a\big)\frac{1}{b-a}\frac{d}{dx}\mathscr{L}_n^i(t)\,dt\\ &= \frac{b-a}{2}\int_{-1}^1 f\Big(\frac{(b-a)s}{2}+\frac{a+b}{2}\Big)\frac{1}{b-a}\frac{d}{dx}\mathscr{L}_n^i\Big(\frac{s+1}{2}\Big)\,ds\\ &\simeq \frac{1}{2}\sum_{i=1}^m A_i f\Big(\frac{(b-a)\zeta_i}{2}+\frac{a+b}{2}\Big)\frac{d}{dx}\mathscr{L}_n^i\Big(\frac{\zeta_i+1}{2}\Big).\end{aligned}$$

In chapter 5 we'll need the following expressions as well:

$$\int_a^b f(x)L_n^i(x)L_n^j(x)\,dx \simeq \frac{b-a}{2}\sum_{i=1}^m A_i f\Big(\frac{(b-a)\zeta_i}{2}+\frac{a+b}{2}\Big)\mathscr{L}_n^i\Big(\frac{\zeta_i+1}{2}\Big)\mathscr{L}_n^j\Big(\frac{\zeta_i+1}{2}\Big)$$

$$\int_a^b f(x)L_n^i(x)\frac{d}{dx}L_n^j(x)\,dx \simeq \frac{1}{2}\sum_{i=1}^m A_i f\Big(\frac{(b-a)\zeta_i}{2}+\frac{a+b}{2}\Big)\mathscr{L}_n^i\Big(\frac{\zeta_i+1}{2}\Big)\frac{d}{dx}\mathscr{L}_n^j\Big(\frac{\zeta_i+1}{2}\Big)$$

$$\int_a^b f(x)\frac{d}{dx}L_n^i(x)\frac{d}{dx}L_n^j(x)\,dx \simeq \frac{1}{4(b-a)}\sum_{i=1}^m A_i f\Big(\frac{(b-a)\zeta_i}{2}+\frac{a+b}{2}\Big)\frac{d}{dx}\mathscr{L}_n^i\Big(\frac{\zeta_i+1}{2}\Big)\frac{d}{dx}\mathscr{L}_n^j\Big(\frac{\zeta_i+1}{2}\Big)$$

Notice that to apply these formulas with Lagrange polynomials of degree n, we have to evaluate the functions

$$\mathscr{L}_n^i(x) \qquad \text{and} \qquad \frac{d}{dx}\mathscr{L}_n^i(x)$$

for $i=0,1,2,\ldots,n$ at m different points given by the Gauss quadrature formulas.

3.5 Taylor-Wingate-Bos (TWB) quadrature

Theorem 3.2 gives the optimal distribution of quadrature points and weights for numerically integrating a function of one variable on an interval. Unfortunately no such definitive procedure exists for integrating a function of two variables over a triangle. Certainly, one may regard the triangle as the image of a square, and the square as the product of two intervals, and thus extend the Gaussian quadrature to a triangle. However, the distribution of the quadrature points obtained this way is far from optimal. The purpose of this section is to introduce one such quadrature scheme that has been developed by Taylor, Wingate and Bos in their paper [12].

We will start with the basic triangle $T_{\mathscr{B}}$ with vertices $(0,0)$, $(1,0)$ and $(0,1)$ and describe a quadrature formula

$$\int_{T_{\mathscr{B}}} f(x,y)\,dx\,dy \simeq \sum_{i=1}^{n} A_i f(x_i,y_i)$$

with strength d, that is, it's exact for all polynomials of degree at most d. In the aforementioned paper, the authors describe a method that uses Newton's method to solve the nonlinear system of algebraic equations for the quadrature weights and locations of the points, symmetry to reduce the complexity of the problem and a cardinal function algorithm to obtain points and weights for different values of d. Table 3.4 shows the number of points needed for different values of d.

d	2	4	5	7	9	11	12	14	16	18	20	21	23	25
n	3	6	10	15	21	28	26	45	55	66	78	91	105	120

Table 3.4: Quadrature strengths and number of points n from [12]

For example, if we want to use a quadrature rule that is exact for all polynomials of degree at most 16, we need to evaluate the function $f(x,y)$ at 55 points. In listing 3.3 we can see the TWB (Taylor, Wingate and Bos) data for strengths 2, 4 and 5.

In chapter 24, Integration on triangles, of [9] Rouben Rostamian also describes the work of Taylor, Wingate and Bos and has made the data available at the book's website [8]. Observe that if we integrate the function $f(x,y)=1$ using these quadrature formulas, we get that

$$\int_{T_{\mathscr{B}}} dx\,dy = 2,$$

that's because Taylor, Wingate and Bos use barycentric coordinates and the triangle with vertices $(-1,-1)$, $(1,-1)$ and $(-1,1)$. Since this triangle has area equal to 2, they scale the weights so that

$$\sum_{i=1}^{n} A_i = 2.$$

```
static twb_qdat quaddata3[] = { /* strength=2  N=3 */
   { 0.1666666666667, 0.6666666666667, 0.6666666666667 },
   { 0.6666666666667, 0.1666666666667, 0.6666666666667 },
   { 0.1666666666667, 0.1666666666667, 0.6666666666667 },
   { 0.0, 0.0, -1.0 }   /* terminator */
};

static twb_qdat quaddata6[] = { /* strength=4  N=6 */
   { 0.0915762135098, 0.0915762135098, 0.2199034873106 },
   { 0.8168475729805, 0.0915762135098, 0.2199034873106 },
   { 0.0915762135098, 0.8168475729805, 0.2199034873106 },
   { 0.1081030181681, 0.4459484909160, 0.4467631793560 },
   { 0.4459484909160, 0.1081030181681, 0.4467631793560 },
   { 0.4459484909160, 0.4459484909160, 0.4467631793560 },
   { 0.0, 0.0, -1.0 }   /* terminator */
};

static twb_qdat quaddata10[] = { /* strength=5  N=10 */
   { 0.0000000000000, 1.0000000000000, 0.0262712099504 },
   { 1.0000000000000, 0.0000000000000, 0.0262716612068 },
   { 0.0000000000000, 0.0000000000000, 0.0274163947600 },
   { 0.2673273531185, 0.6728199218710, 0.2348383865823 },
   { 0.6728175529461, 0.2673288599482, 0.2348412238268 },
   { 0.0649236350054, 0.6716530111494, 0.2480251793114 },
   { 0.6716498539042, 0.0649251690029, 0.2480304922521 },
   { 0.0654032456800, 0.2693789366453, 0.2518604605529 },
   { 0.2693767069140, 0.0654054874919, 0.2518660533658 },
   { 0.3386738503896, 0.3386799893027, 0.4505789381914 },
   { 0.0, 0.0, -1.0 }   /* terminator */
};
```

Source code 3.3: Snippet of TWB quadrature data for different values of d and n from [8]

We have to take this fact into account when we write the C code to compute integrals over triangles. Since our base triangle has area 0.5, we have to scale the weights by 4.

Finally, we will represent these quadrature points with (ζ_i, η_i) and to compute an integral over a triangle T with vertices (x_0, y_0), (x_1, y_1) and (x_2, y_2)

$$\int_T f(t,s)\,dt\,ds$$

we can do the change of variable defined in equation 3.2, and obtain

$$\int_T f(t,s)\,dt\,ds = \int_{\mathscr{B}} f\left(\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}\right)\left|\begin{matrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{matrix}\right| dx\,dy$$

$$= 2|T| \int_{\mathscr{B}} f\left(\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}\right) dx\,dy$$

$$= \frac{|T|}{2} \sum_{i=1}^{n} A_i f\left(\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{pmatrix}\begin{pmatrix} \zeta_i \\ \eta_i \end{pmatrix}\right)$$

CHAPTER 4

Two dimensional domains

The finite element method works by solving the differential equation in small discrete elements, which when grouped together amount to the whole domain. The definition of these elements will be different according to the different amount of dimensions the domain has. Then, the first step is to describe the domain Ω. For one-dimensional problems it is easy: the domain is an interval and we need only the endpoints.

A two-dimensional domain can be defined in different ways, for example, describing it by a set of inequalities or describing its boundary with different curves in parametric form. Like many books, we shall approximate the boundary of the domain with a set of polygonal lines. If we have a simple domain that can be described with few points, then we can list all of them in the code of the program.

For example, in chapter 23 of book [9], the author describes a domain that consists of a triangle with a hole and can be seen in figure 4.1 in the code of the main program, see the listing 4.1

But if the number of points to approximate the boundary becomes very large, this method becomes impractical, hence the need for other methods to describe more complex situations. In fact, the strength of the Finite Element Method is its ease of handling domains with complex shapes. After considering different options, we have decided to use the XML language to describe the two-dimensional domains and the boundary conditions.

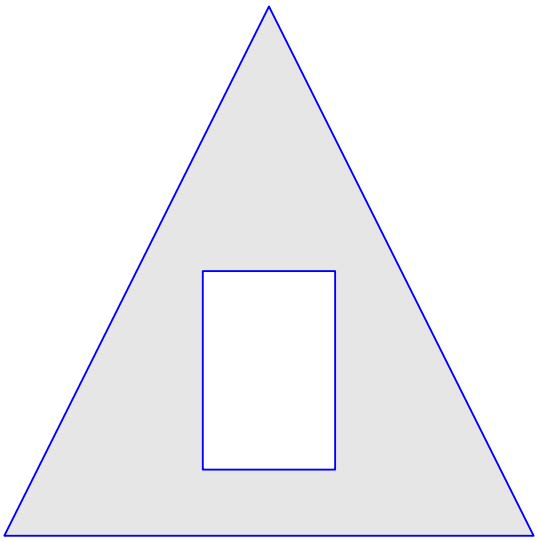

Figure 4.1: Triangle with a hole

```
1  struct problem_spec *triangle_with_hole(void)
2  {
3     static struct problem_spec_point points[] = { // the points
4       // triangle's vertices
5       { 0, -1.0, 0.0, FEM_BC_DIRICHLET },
6       { 1,  1.0, 0.0, FEM_BC_DIRICHLET },
7       { 2,  0.0, 2.0, FEM_BC_DIRICHLET },
8       // hole's vertices
9       { 3, -0.25, 0.25, FEM_BC_DIRICHLET },
10      { 4,  0.25, 0.25, FEM_BC_DIRICHLET },
11      { 5,  0.25, 1.0 , FEM_BC_DIRICHLET },
12      { 6, -0.25, 1.0 , FEM_BC_DIRICHLET }
13    };
14
15    static struct problem_spec_segment segments[] = { // the segments
16      // triangle's segments
17      { 0, 0, 1, FEM_BC_DIRICHLET },
18      { 1, 1, 2, FEM_BC_DIRICHLET },
19      { 2, 2, 0, FEM_BC_DIRICHLET },
20      // hole's segments
21      { 3, 3, 4, FEM_BC_DIRICHLET },
22      { 4, 4, 5, FEM_BC_DIRICHLET },
23      { 5, 5, 6, FEM_BC_DIRICHLET },
24      { 6, 6, 3, FEM_BC_DIRICHLET },
25    };
26
27    static struct problem_spec_hole holes[] = { // the hole's identifier
28      { 0.0, 0.75 }
29    };
```

Source code 4.1: Code description of a simple domain from [9]

4.1 XML description of a domain

According the World Wide Web Consortium (W3C) [14], Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally

designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. Typical examples of XML are Microsoft and Open Office files[1], HTML for web design and SVG for vector images.

The XML document represents a structure which involves tag and entities and elements that are a subset of these entities. Each element describes one or more attributes and attributes define the method to process this element. Any file created in XML can be read, processed, and written in any operating system so that you can easily transfer and exchange XML files between different platforms.

The pros of this format are:

- They are self-documenting: It's easy to interpret what each part means. They also support comments.
- They are an industry standard. This means that there are many tools to work with them and much documentation on proper syntax.
- They are scalable. One could easily add new tags to add functionalities to the program.

To parse the XML documents that we use in our project, we use the *The XML C parser and toolkit of Gnome* [13]. In the project's website, we can see that Libxml2 is the XML C parser and toolkit developed for the Gnome project (but usable outside of the Gnome platform), it is free software available under the MIT License. XML itself is a metalanguage to design markup languages, i.e. text language where semantic and structure are added to the content using extra "markup" information enclosed between angle brackets. HTML is the most well-known markup language. Though the library is written in C a variety of language bindings make it available in other environments.

Then, we can define the simple domain of a triangle with a hole with the XML listed in 4.2. The root element of the XML is `<domain-2d>` and has children of type `<region>` and `<hole>`.

The basic shapes of our XML format are polygons, and ellipses. Ellipses can be rotated around their center. We can also have closed paths that can be defined with lists of points, Bézier and parametric curves. We shall not describe all the tags and options that we can use in the XML files to describe the domain, they can be seen in the following examples.

In the first example, we see the use of ellipses and different types of boundary conditions. We can see the XML code in the listing 4.3 and the domain in the figure 4.2.

[1]These are actually zip files, however all the information is held in XML files inside these zips. One can explore them by changing their extension to .zip and opening them.

```
<?xml version="1.0" encoding="UTF-8"?>
<regions>
  <region type="polygon" bc="2">
      <point x="-1" y="0" />
      <point x="1" y="0" />
      <point x="0" y="2" />
  </region>
  <region type="polygon" bc="2">
      <point x="-0.25" y="0.25" />
      <point x="0.25" y="0.25" />
      <point x="0.25" y="1" />
      <point x="-0.25" y="1" />
  </region>
  <holes>
      <hole x="0.0" y="0.75"/>
  </holes>
</regions>
```

Source code 4.2: XML description of a simple domain

```
<?xml version="1.0" encoding="UTF-8"?>
<domain-2d>
  <region type="ellipse" bc="4">
    <center x="0" y="0" ax="3.0" ay="8.0" segments="36" rotate="45"/>
  </region>
  <region type="ellipse" bc="3" >
    <center x="-2" y="2" ax="2" ay="1.0" segments="36" rotate="45" />
  </region>
  <region type="ellipse" bc="2">
    <center x="2" y="-2" ax="2" ay="1.0" segments="36" rotate="45" />
  </region>
  <hole x="-2"  y="2" />
  <hole x="2"  y="-2" />
</domain-2d>
```

Source code 4.3: XML description of a second domain

In the second example, we can see a domain defined by a path with two parametric curves, see listing 4.4 and figure 4.3

4.2 Delaunay Triangulation

Once we have the domain defined, we have the necessity to break it down into small, finitely sized elements (hence Finite Element Method). The smaller these are, the closer to a continuous medium we'll be, and the more accurate the solution.

Although other shapes are capable of tesselation (rectangles, hexagons), triangles are by far the most used. The goal of a triangulation is to generate a mesh of triangles within a domain such that:

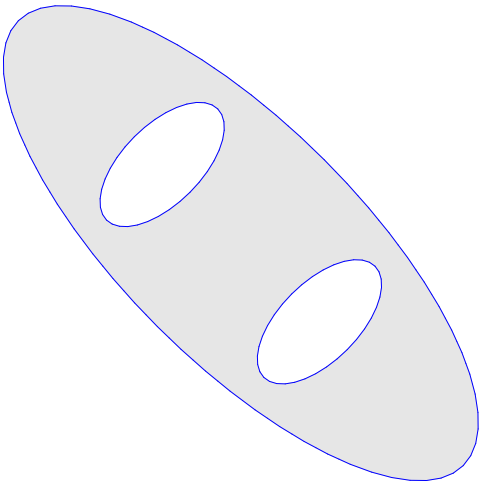

Figure 4.2: Domain with three ellipses

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <domain-2d>
4     <region type="path">
5        <curve x="t" y="sqrt(t^3*(4-t))/2" segments="40" begin="0" end="4" bc="2" />
6        <point x="4" y="0" bc="3" />
7        <curve x="t" y="-sqrt(t^3*(4-t))/2" segments="40" begin="4" end="0" bc="3"/>
8     </region>
9  </domain-2d>
```

Source code 4.4: Domain defined by curves in parametric form

- The entire domain is covered by triangles.
- No two triangles overlap
- No triangle vertex lies on another triangle's edge.
- Triangles are not excessively elongated: their smallest angle should be maximized
- No triangle is larger than a given maximal area

Actually developing a working program to triangulate two or three dimensional domains is a difficult task. There are many algorithms to do so. Fortunately, there is a free library developed by J. Shewchuk, called Triangle and self described as *The Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. We can simply outsource the job of triangulating to this library whilst our solver does the rest, though we must still understand how it works.

The aforementioned library uses Delaunay triangulation, a method developed by Boris Delaunay. This method is very popular because it minimizes triangle maximal area

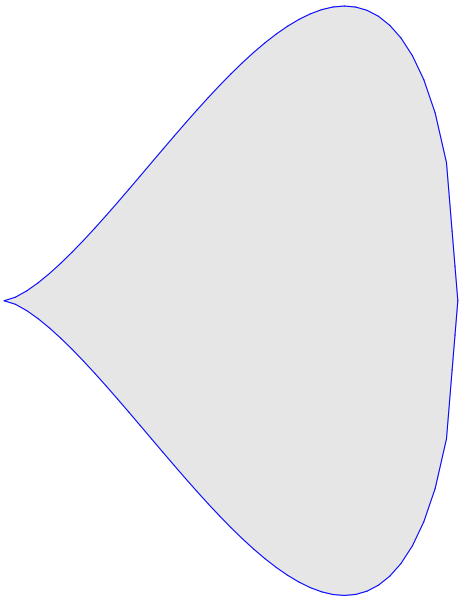

Figure 4.3: First domain defined by curves in parametric form

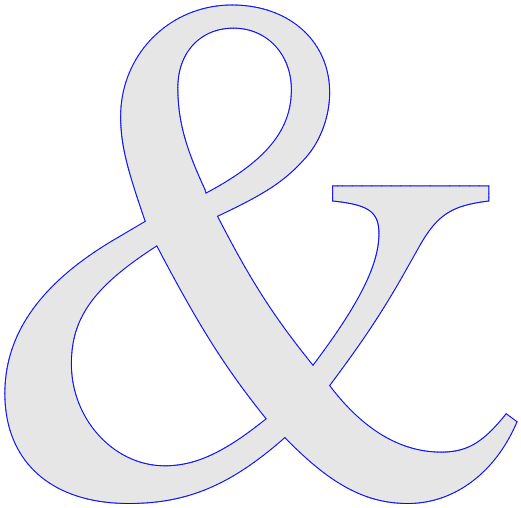

Figure 4.4: Domain formed with Bézier curves

and maximizes the minimum vertex angle, both properties that enhance the precision of the FEM.

As an addendum, it must be mentioned that a Delaunay triangulation is not unique for a domain, different refinement algorithms will yield different resulting triangulations.

```
<?xml version="1.0" encoding="UTF-8"?>
<regions>
 <region type="path">
  <point x="12.8" y="8.3999998"/>
  <point x="8.6687602" y="8.3999998"/>
  <point x="8.6687602" y="7.9999998"/>
  <bezier segments="36">
    <point x="8.6687602" y="7.9999998"/>
    <point x="9.6250004" y="7.9031402"/>
    <point x="9.8937602" y="7.7124998"/>
    <point x="9.8937602" y="7.1374998"/>
  </bezier>
  <bezier segments="36">
    <point x="9.8937602" y="7.1374998"/>
    <point x="9.8937602" y="6.3156402"/>
    <point x="9.3750004" y="5.2812598"/>
    <point x="8.1500004" y="3.6562602"/>
  </bezier>
  <bezier segments="36">
    <point x="8.1500004" y="3.6562602"/>
    <point x="7.0812602" y="4.9749998"/>
    <point x="6.4687602" y="5.9531402"/>
    <point x="5.6250004" y="7.5968802"/>
  </bezier>

  . . . . . .

  <bezier segments="36">
    <point x="1.75938" y="3.6937602"/>
    <point x="1.75938" y="4.9187598"/>
    <point x="2.3531398" y="5.7406402"/>
    <point x="4.0187602" y="6.8124998"/>
  </bezier>
 </region>
 <hole x="3.8406398" y="4.39298"/>
 <hole x="5.5414204" y="11.192979"/>
</regions>
```

Source code 4.5: Second domain defined by a list of points and Bézier curves

4.3 Voronoi tesselation

Also known as Thiessen tessellation of Dirichlet tessellation (the same Dirichlet that defined the Boundary Conditions), it's a way to tesselate a 2D domain. It creates irregular shapes with unequal number of sides, so it's not ideal for the FEM. It's useful, however, because it is the dual graph of a Delaunay triangulation. Let's expand on this.

A Voronoi tesselation is easy to understand: the domain is divided according to which point is the closest. This leads to an irregular division of space, with one node in each region. If two regions share a border, we say they are neighbours.

We say they are dual graphs because each Voronoi border is perpendicularly intersected by one edge of the triangulation. Sometimes they intersect with the extension of the border instead of the border itself (e.g: figure 4.5 edge A-E). In other words, only nodes in neighbouring areas are connected by the triangulation. This can be seen in figure 4.5.

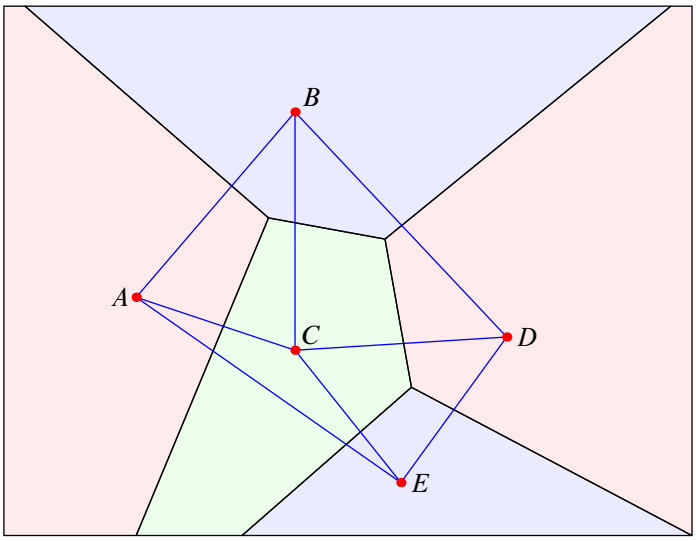

Figure 4.5: A Voronoi map and Delaunay triangluation

4.4 Meshing

First of all, a mesh of all nodes is created. Here an important distinction must be made. What we need is not simply a Delaunay triangulation, but a *constrained Delaunay triangulation*. This means that the edges of our domain must be the edges of triangles. This can be seen in figure 4.6.

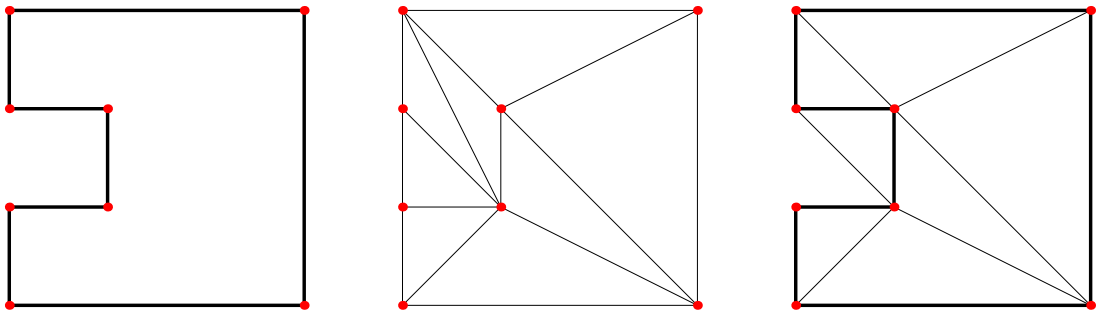

Figure 4.6: A domain, a triangulation and a constrained triangulation

Once the first constrained mesh is done, all triangles lying on the outside the domain (either the proper outside or within a hole) are removed, leaving us with an unrefined mesh (figure 4.7).

This triangulation, however, does not yet guarantee the Delaunay triangulation properties exposed earlier. For them to be fulfilled, the mesh must be refined.

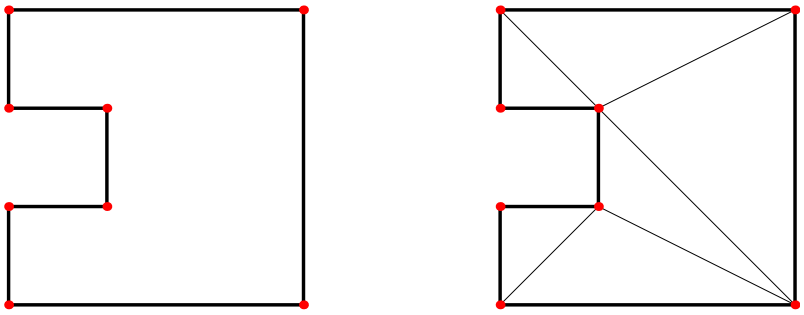

Figure 4.7: A two dimensional domain and a constrained, unrefined triangulation

4.5 Refinement

The chosen library works recursively, by fixing bad segments and bad triangles until there are none more of them. To determine which of these elements are bad it uses two different yet similar criteria:

Any *segment* has its diametral circle, the smallest circle that passes through both its endpoints. The segment is said to be *encroached* if its diametral circle encompasses another node. To fix it, a new node is created at the midpoint of the segment, and the segment is split in half.

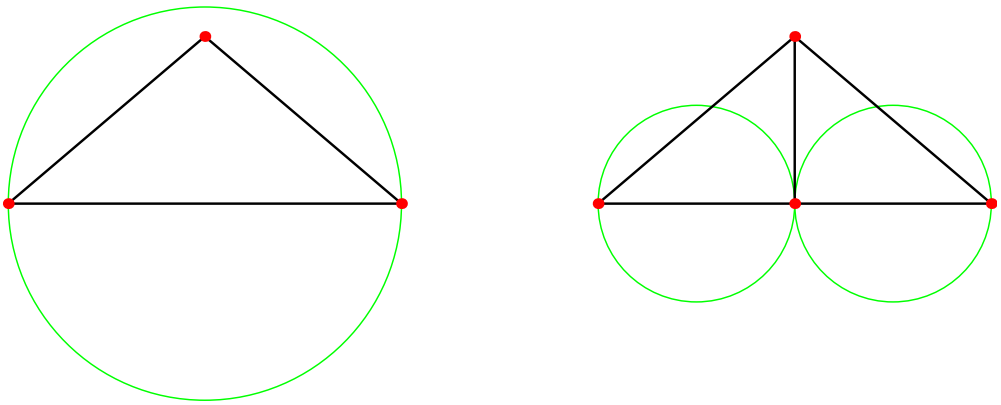

Figure 4.8: An encroached segment is fixed

This does not always guarantee that none of the newly created segments will become encroached themselves. Should that happen, this new shorter encroached segment would be added to the queue of encroached segments.

Similarly, a *triangle* is said to be *bad* if either it's bigger than the maximum acceptable area or its circumcircle encompasses another node. To fix it, a new node is created at

the circumcenter, and connected to the three vertices, hence splitting the triangle into three.

There exists the possibility that this new central node encroaches on some of the edges of the triangle. If that happens, the new circumcentral node is deleted, and all edges that would have been encroached upon are split in half. This is shown in figure 4.9, where the first attempt to fix the bad triangle causes an encroachment on the red edge, thus the process is reversed and the segment split.

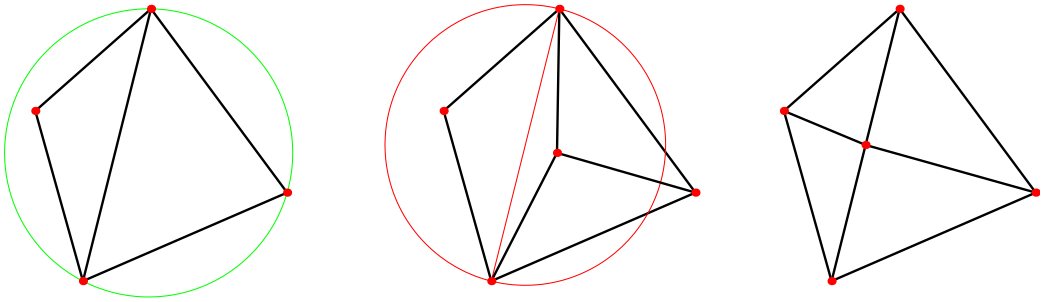

Figure 4.9: A bad triangle is fixed

Encroached segments are prioritized over bad triangles. The program sets up two queues and solves segments first and triangles later. Every time an element it fixed, new elements might be added to each queue. The order in which it solves each queue is first-come first-serve.

More detailed information on the algortithm can be obtained from [1] and specially the library's author's paper: [10].

4.6 Storing the triangulation

When assembling the stiffness matrix, the program will need to read information on the properties and relationships of each element, edge and node. In reality, the program only needs to access the information downwards, that is, you might want to know what nodes belong to a particular triangle, but at no point do you need to know which triangle a particular node belongs to.

The solution goes as follows. There is an object, the Mesh, which has arrays containing the nodes, edges and elements (triangles), as well as some meta-information. The elements themselves have pointers to their nodes and edges, and the edges have pointers to their nodes. This can be seen in listing 4.6.

This listing also shows a couple of functions. The first one is a wrapper around the Triangle library, and the second simply frees all pointers in the mesh.

```c
typedef struct{
   int nodeno;
   double x;
   double y;
   double z;
   int bc;

   int boundaryno;
} node;

typedef struct{
   int edgeno;
   node *n[2];
   int bc;

   int *Points;
   int *Points_boundary;
} edge;

typedef struct{
   int elemno;
   node *n[3];
   edge *e[3];
   double ex[3], ey[3];
   double area;

   int **Points;
} elem;

typedef struct{
   node *nodes;
   edge *edges;
   elem *elems;

   int nnodes;
   int nedges;
   int nelems;

   int npoints;
   int n_boundary_points;
} mesh;

mesh *make_mesh (problem_spec *spec, double a);
void free_mesh(mesh *mesh);
```

Source code 4.6: C objects that describe the mesh

Notice that the declaration seems to conflate two naming patterns: nodes vs. points. It is in fact not an inconsistency. The nodes are the subset of points that lie at the vertex of a triangle. When interpolating with polynomials of degree 1, all nodes are points, but higher degree interpolations require extra points. Their Cartesian coordinates can be learned from the nodes, so there is no need to store them. What we need to store is the global coordinates of these points, hence the arrays `Points`. It works such that `Mesh.elems[e].Points[i][j]` stores the global coordinate of the point belonging to triangle e with local coordinates (i, j).

This is better understood with figure 4.14 in mind. In this figure we can see how border points have two coordinates: one related to u and the other one to $\dot{u}$. The global coor-

dinates relating to $\dot{u}$ have the word `boundary` in them. For instance `int boundaryno` or `int *Points_boundary`.

Finally, we will see some examples of triangulations. In the first example, the domain is the same that we defined in the XML file in 4.4. The domain triangulated can be seen in figure 4.10.

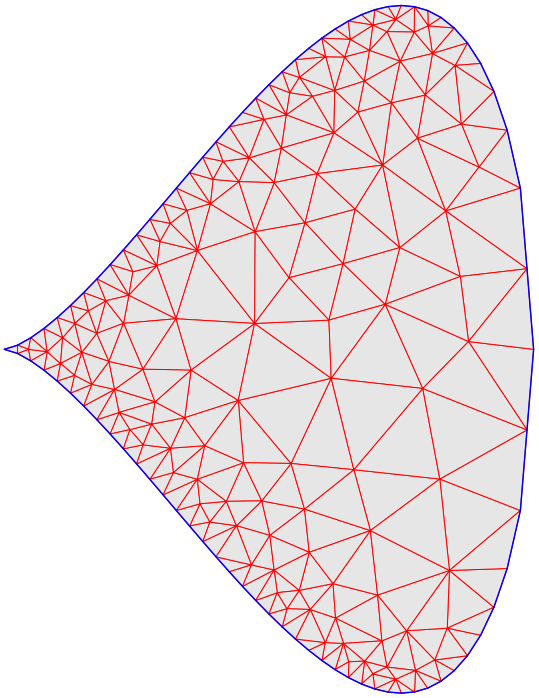

Figure 4.10: Triangulation of a domain defined by parametric curves

The second example is an L-shaped domain with three holes. The XML code to describe the domain is in listing 4.7 and we can find its triangulation in figure 4.11

In figure 4.12, we can see the mesh with the triangles and the nodes when we use interpolating polynomials of degree four. In this case, the mesh consists in 310 vertices, 792 edges and 481 triangles. Moreover, 141 of these vertices belong to the boundary and also 141 edges are on the boundary. Since we want to use fourth degree polynomials to interpolate, we have 4129 points, meaning 4129 unknowns in the system of equations (the values of the unknown function $u(x, y)$ at each of these points).

Although that's not all, 564 of the nodes are boundary nodes, and for each boundary node, we have another unknown, the value of the normal derivative of the function $u(x, y)$. Then, to solve a FEM problem in this domain and triangulation, we have to solve a system of equations with 4693 unknowns. In figure 4.13, we can see a close-up of the previous domain and triangulation.

```
<?xml version="1.0" encoding="UTF-8"?>
<regions>
 <region type="polygon">
   <point x="0" y="0" />
   <point x="2" y="0" />
   <point x="2" y="2" />
   <point x="4" y="2" />
   <point x="4" y="4" />
   <point x="0" y="4" />
 </region>
 <region type="circle" bc="4">
   <center x="1" y="1" radius="0.325" segments="24" />
 </region>
 <region type="circle">
   <center x="1" y="3" radius="0.325" segments="24" />
 </region>
 <region type="circle" bc="3">
   <center x="3" y="3" radius="0.325" segments="24" />
 </region>
 <hole x="1" y="1"/>
 <hole x="1" y="3"/>
 <hole x="3" y="3"/>
</regions>
```

Source code 4.7: An L-shaped domain with three holes

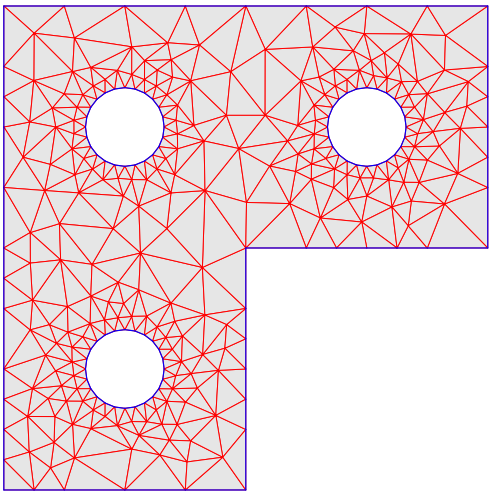

Figure 4.11: Triangulation of an L-shaped domain with holes

To compute the matrix and solve this system of equations, we need to have the unknowns labeled or numbered, from the first unknown to the last one. In the process of triangulation of a domain, we obtain an ordered list of the points (vertices of the triangles), an ordered list of the edges and an ordered list of the triangles, then the criterion that we have used to sort the unknowns is:

- The first unknowns are the values of the function u at the vertices of the triangles.

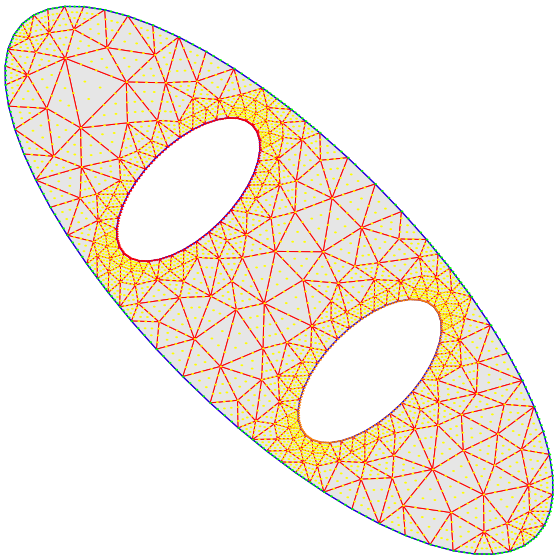

Figure 4.12: Triangulation and nodes of a mesh

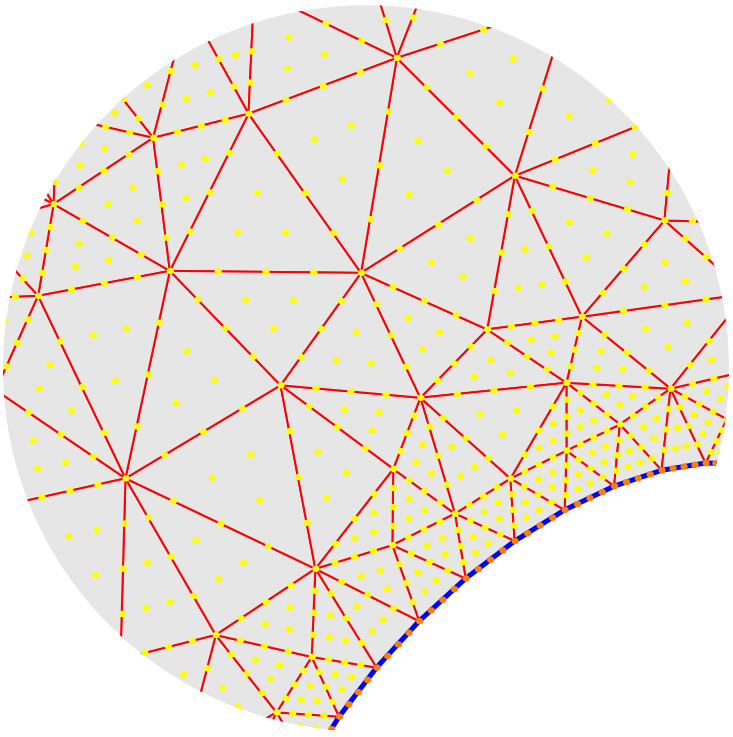

Figure 4.13: Detail of a triangulation and nodes of a mesh

- If the degree of the interpolation polynomials is greater than one, $d > 1$, then the inner points of the edges. We have $d-1$ points for every edge.

- If the degree of the interpolation polynomials is greater than two, $d > 2$, the inner points of every triangle. In this case, we have

$$\frac{(d-1)(d-2)}{2}$$

  inner points in each triangle.

- The next unknowns are the values of the normal derivative of the function u at the boundary points.
- Finally, the values of the normal derivative of the function u at the interior points of the boundary edges.

In figure 4.14, we can see the unknowns of a rectangular domain with only four triangles and interpolation polynomials of degree 5.

In this case, we have 6 nodes (all boundary points), 9 edges (6 of them on the boundary) and 4 triangles. Then the total number of unknowns is

$$6 + 9 \cdot 4 + \frac{4 \cdot 4 \cdot 3}{2} + 6 + 6 \cdot 4 = 96 .$$

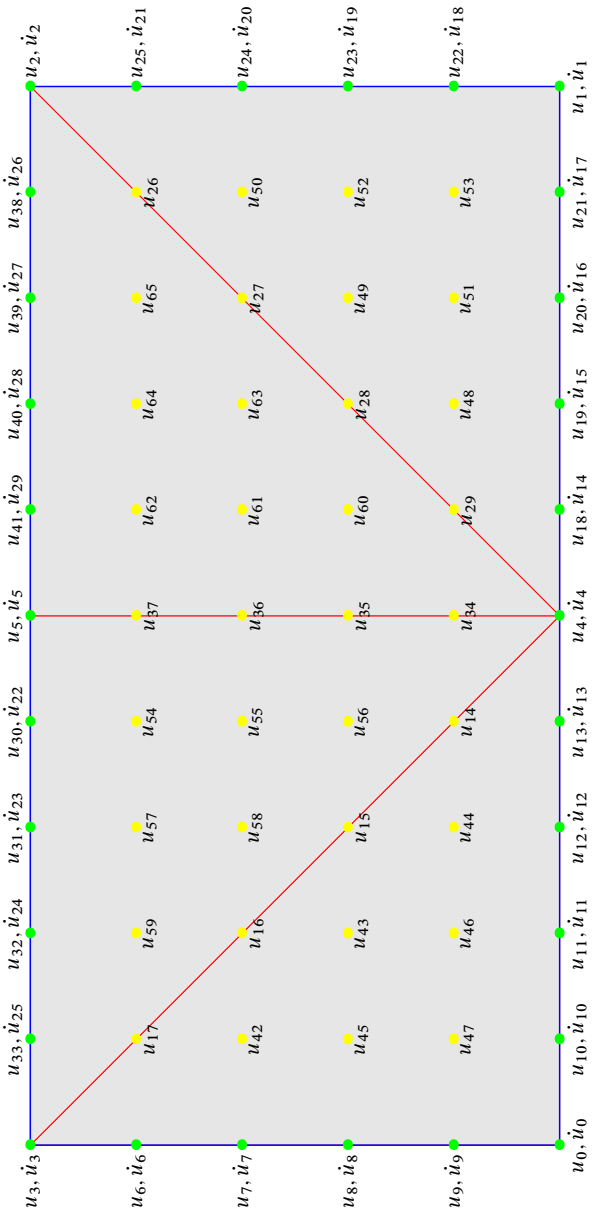

Figure 4.14: Unknowns for a rectangular domain with four triangles

CHAPTER 5

The finite element method in dimension one

5.1 Introduction

One dimensional problems are not very frequent in physics. One context in which they appear is when there is some symmetry in the system that allows us to ignore the other dimensions. For instance, the example of the adiabatic tube exposed in section 2.3. Similar examples include spheres where heat is generated (point symmetry) or cylindrical resistors with an internal heat source (axial symmetry). They can also arise when the dimensions of an object in one direction are much longer than the other two. The classical example is the torsion or bending of a beam.

5.2 Historical background

Indeed it is the bending of a beam that first motivated the development of the FEM. Hrennikoff in 1941 and McHenry in 1943 developed methods to divide bars and beams into smaller discrete elements. The lack of computational power at the time meant that these still had to be solved analytically, but it was a first step in the right direction. These methods were used mostly in the field of aeronautics, specially in structural design.

Aerospace engineering outpaced these methods, however. These where found to be impractical when jets first went supersonic. Throughout the 1950's, in the pinnacle of the cold war, and with more computational power becoming available, Argyris

and Kesley proved to be able to solve complicated problems using computational techniques. In 1956 the stiffness matrix was born out of a paper by Turner *et al.*. They also described how to assemble it. Finally in 1960, Clough first introduced the terms Finite Elements. Soon after the first 2D problems started to be solved, but that's a story for a later time [15].

5.3 Weak form of the problem and the discrete system

Despite our main goal being to develop a solver for the two-dimensional FEM, we must take it one step at a time. Our first important milestone will be to develop a functional solver for one-dimensional problems. This will serve us to ascertain that the previously explored mathematical tools have been properly implemented. Furthermore, there are many analytical solutions for 1D problems, which will allow us to check whether the problem works, or, alternatively, narrow down where the error may be.

Although most subroutines in the two dimension solver will be different, most of them can be understood by analogy to these. In order to learn the FEM, programming the 1D version first allows one two divide their learning. Once a functional 1D solver is written, the student will understand Gauss quadrature, Laplace interpolation, sparse matrices and assembly of the system. These are concepts that are used in the 2D version as well, even if they become a bit more complex.

The mathematical development will not be completely rigorous, since we will not define the exact properties the solution and the domain must have. In general, they have to be *nice*: no discontinuities, piecewise differentiable, etc. For a more rigorous approach one may refer to [11].

The problem we'll solve is the following:

$$-\frac{d}{dx}\left(a_2(x)\frac{d}{dx}u(x)\right)+a_1(x)\frac{d}{dx}u(x)+a_0(x)u(x)=f(x) \tag{5.1}$$

The weak form of the equation, also called variational formulation, is that which we can turn into a linear equation system. To obtain it, we start by integrating both sides with the help of a test function $\varphi(x)$.

$$\int_a^b\left(-\frac{d}{dx}\left(a_2(x)\frac{d}{dx}u(x)\right)+a_1(x)\frac{d}{dx}u(x)+a_0(x)u(x)\right)\varphi(x)dx=\int_a^b f(x)\varphi(x)dx \tag{5.2}$$

This equation must hold for all $\varphi(x)\in V$. Without getting too technical, V is a function space that must fulfill a certain set of properties, in fact a Sobolev space, where $u(x)$ and $\varphi(x)$ must belong. Checking whether this holds for all $\varphi(x)\in V$ is impossible, since there are infinitely many functions, so we will use a sufficiently large subspace of V. As

we have seen in chapter 3, we will use the finite dimensional subspace $V' = \mathbb{P}^n_m([a,b])$, i. e., the continuous functions in $[a,b]$ such that their restriction to I_k, $k = 0,1,\ldots,m-1$ is a polynomial of degree at most n. This is called the test space.

Suppose now that $\varphi_i(x)$ for $i = 0,1,\ldots,mn$ is the basis of $\mathbb{P}^n_m([a,b])$ described in chapter 3, then we want to obtain a function $u(x)$ such that

$$\int_a^b \left(-\frac{d}{dx}\left(a_2(x)\frac{d}{dx}u(x)\right)+a_1(x)\frac{d}{dx}u(x)+a_0(x)u(x)\right)\varphi_i(x)dx = \int_a^b f(x)\varphi_i(x)dx \tag{5.3}$$

for all $i = 0,1,\ldots,mn$.

Let's now explore the first term on its own. We can use integration by parts to obtain the following result:

$$\int_a^b \frac{d}{dx}\left(a_2(x)\frac{d}{dx}u(x)\right)\varphi_i(x)dx =$$

$$= \left[a_2(x)\frac{d}{dx}u(x)\varphi_i(x)\right]_a^b - \int_a^b a_1(x)\frac{d}{dx}u(x)\frac{d}{dx}\varphi_i(x)dx$$

Having done this, we can now rearrange equation 5.3 so that it becomes

$$\begin{aligned}\int_a^b a_2(x)\frac{d}{dx}u(x)\frac{d}{dx}\varphi_i(x)dx + \int_a^b a_1(x)\frac{d}{dx}u(x)\varphi_i(x)dx \\ + \int_a^b a_0(x)u(x)\varphi_i(x)dx - \left[a_2(x)\frac{du}{dx}(x)\varphi_i(x)\right]_a^b \\ = \int_a^b f(x)\varphi_i(x)dx\end{aligned} \tag{5.4}$$

Until now, we have replaced the original equation 5.1 with $mn+1$ equations, one for every basis function φ_i; so when we see a φ_i in an equation, that means that this is the equation number i. This is the first discretization step in the finite element method.

In the second discretization step, we rewrite the problem saying that we want to find a function $u \in \hat{V}$, where $\hat{V}$ is a finite dimensional space of functions, such that 5.3 holds for all $i = 0,1,\ldots,mn$. $\hat{V}$ is called the trial space. Although the test and the trial space can be different spaces, normally they are chosen to be the same, in our case $V' = \hat{V} = \mathbb{P}^n_m([a,b])$. Then, we can approximate the unknown $u(x)$ and its derivative

by

$$u(x) = \sum_{j=0}^{N} u_j \varphi_j(x)$$

$$\frac{d}{dx} u(x) = \sum_{j=0}^{N} u_j \frac{d}{dx} \varphi_j(x),$$

where $N = mn$. At this moment we have $N+1$ equations with $N+1$ unknowns. Let's develop the integrals on the left side of equations 5.4. For the first integral, we have that

$$\begin{aligned}\int_a^b a_2(x) \frac{d}{dx} u(x) \frac{d}{dx} \varphi_i(x) dx &= \int_a^b a_2(x) \left(\sum_{j=0}^{N} u_j \frac{d}{dx} \varphi_j(x) \right) \frac{d}{dx} \varphi_i(x) dx \\ &= \sum_{j=0}^{N} u_j \int_a^b a_2(x) \frac{d}{dx} \varphi_j(x) \frac{d}{dx} \varphi_i(x) dx \\ &= \sum_{j=0}^{N} k_{i,j}^1 u_j,\end{aligned}$$

where

$$k_{i,j}^1 = \int_a^b a_2(x) \frac{d}{dx} \varphi_j(x) \frac{d}{dx} \varphi_i(x) dx.$$

The second integral is

$$\begin{aligned}\int_a^b a_1(x) \frac{d}{dx} u(x) \varphi_i(x) dx &= \int_a^b a_1(x) \left(\sum_{j=0}^{N} u_j \frac{d}{dx} \varphi_j(x) \right) \varphi_i(x) dx \\ &= \sum_{j=0}^{N} u_j \int_a^b a_1(x) \frac{d}{dx} \varphi_j(x) \varphi_i(x) dx \\ &= \sum_{j=0}^{N} k_{i,j}^2 u_j,\end{aligned}$$

where

$$k_{i,j}^2 = \int_a^b a_1(x) \frac{d}{dx} \varphi_j(x) \varphi_i(x) dx.$$

The third integral is

$$\begin{aligned}\int_a^b a_0(x)u(x)\varphi_i(x)dx &= \int_a^b a_0(x)\left(\sum_{j=0}^{N} u_j\varphi_j(x)\right)\varphi_i(x)dx \\ &= \sum_{j=0}^{N} u_j \int_a^b a_0(x)\varphi_j(x)\varphi_i(x)dx \\ &= \sum_{j=0}^{N} k^3_{i,j}u_j,\end{aligned}$$

where

$$k^3_{i,j} = \int_a^b a_0(x)\varphi_j(x)\varphi_i(x)dx\,.$$

The last term in the left side of the equation 5.4 is

$$-\left[a_2(x)\frac{d}{dx}u(x)\varphi_i(x)\right]_a^b = -a_2(b)u'(b)\varphi_i(b) + a_2(a)u'(a)\varphi_i(a),$$

but, by the definition of the function $\varphi_i(x)$,

$$\varphi_i(a) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad \varphi_i(b) = \begin{cases} 1 & \text{if } i = N \\ 0 & \text{otherwise}\,. \end{cases}$$

Then, in the first equation ($i = 0$), we have a new term, $a_2(a)u'(a)$ and, in the last equation, we have the term $-a_2(b)u'(b)$, where $u'(a)$ and $u'(b)$ are new unknowns of our system.

The right term of the equations 5.4, called the load (or sometimes force) vector is

$$f_i = \int_a^b f(x)\varphi_i(x)dx \qquad i = 0, 1, \ldots, N$$

Thus, we have $N+1$ equations with $N+3$ unknowns, $u_0, u_1, \ldots, u_N, u'(a), u'(b)$. The last two equations come from the boundary conditions and can have different forms depending on the type of boundary condition. $u_0 = A$ and $u_N = B$ for Dirichlet BC, $u'(a) = A$ and $u'(b) = B$ for Neumann BC and $A_1 u_0 + B_1 u'(a) = C_1$ and $A_2 u_N + B_2 u'(b) = C_2$ for Robin BC.

The equation we obtained before can be written as a traditional $KU = F$ linear system, and then we can use a solver to obtain U. For better understanding, it's useful to have a map of the matrices, vectors and sub-matrices of the system. This is shown in figure 5.1.

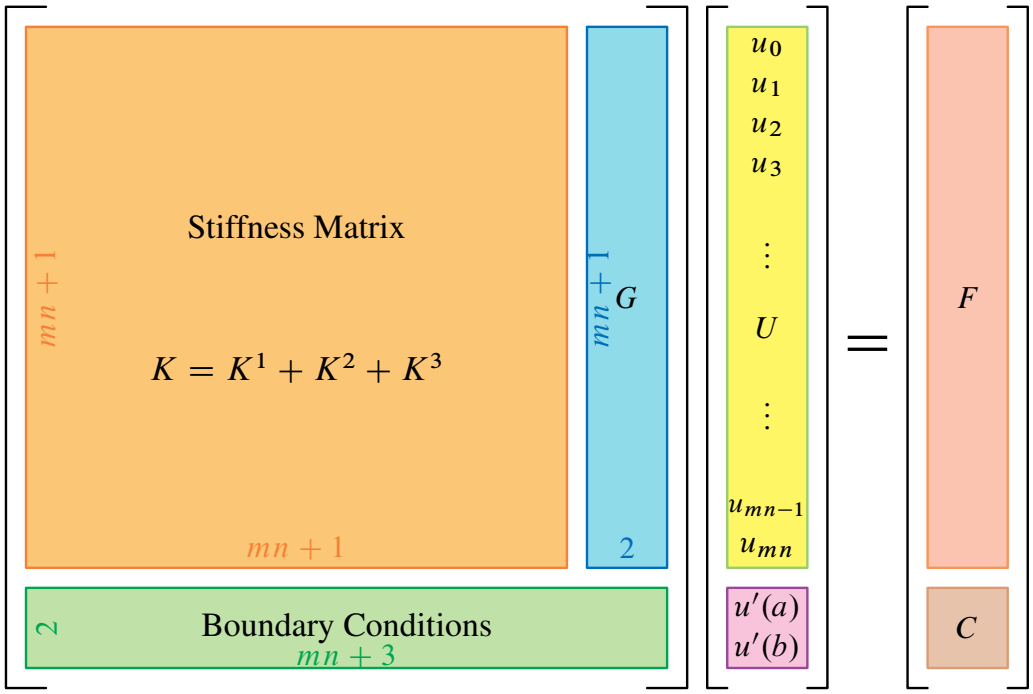

Figure 5.1: Map of the linear system of equations for a 1D domain

5.4 Computation of K and F

To compute the matrices K and F, we need to compute a lot of integrals, namely

$$
\begin{aligned}
k^1_{i,j} &= \int_a^b a_2(x)\frac{d}{dx}\varphi_j(x)\frac{d}{dx}\varphi_i(x)dx \\
k^2_{i,j} &= \int_a^b a_1(x)\frac{d}{dx}\varphi_j(x)\varphi_i(x)dx \\
k^3_{i,j} &= \int_a^b a_0(x)\varphi_j(x)\varphi_i(x)dx \\
f_i &= \int_a^b f(x)\varphi_i(x)dx
\end{aligned}
\tag{5.5}
$$

and we need to compute them efficiently. Remember that the interval $[a,b]$ is divided in m subintervals of the same length, $I_0, I_1, \ldots, I_{m-1}$ and that, fixed the interval I_k, only the functions $\varphi_{kn}(x), \varphi_{kn+1}(x), \ldots, \varphi_{(k+1)n}(x)$ and their derivatives are non-zero in this interval. Moreover, $k^1_{i,j}$ and $k^3_{i,j}$ are symmetrical, in other words,

$$k^1_{i,j} = k^1_{j,i} \qquad \text{and} \qquad k^3_{i,j} = k^3_{j,i}$$

for all $i, j = 0, 1, \ldots, N$, then it's enough that we compute only one of them.

```
1  double *assembleForceVector(double a, double b, int intervals,
2                 int degree, void *f, gauss_qdat qdat, Lagrange lv){
3     double *F;
4     make_vector(F,intervals * degree + 3);
5
6     double xa, xb, dx;
7     xa = a;
8     dx = (b - a)/intervals;
9
10    for(int s=0; s<intervals; s++){ //Looping through intervals
11       xb = xa + dx;
12
13       for(int i=0; i<=degree; i++){ //Looping through points
14          int global = degree*s + i;
15          F[global] += get_Fi(xa, xb, i,f, qdat, lv);
16       }
17       xa = xb;
18    }
19    return F;
20 }
```

Source code 5.1: Loops to obtain the load vector in 1D

Finally, only the functions $\varphi_n(x), \varphi_{2n}(x), \ldots, \varphi_{(m-1)n}(x)$ and their derivatives are non-zero in two of the subintervals $I_0, I_1, \ldots, I_{m-1}$, all other basis functions are non-zero in a single subinterval. For instance, to compute $k^1_{2n,2n}$, we have to compute two integrals

$$k^1_{2n,2n} = \int_{I_0} a_2(x)\frac{d}{dx}\varphi_{2n}(x)\frac{d}{dx}\varphi_{2n}(x)dx + \int_{I_1} a_2(x)\frac{d}{dx}\varphi_{2n}(x)\frac{d}{dx}\varphi_{2n}(x)dx$$

and

$$f_{2n} = \int_{I_0} f(x)\varphi_{2n}(x) + \int_{I_1} f(x)\varphi_{2n}(x),$$

but in most cases we have to compute only one integral over some I_k. Taking into account these facts, in listings 5.2 and 5.1 we can find the loops needed for the computation of the load vector and the stiffness matrix. Note that we store the values in a triplet form matrix.

All integrals are computed using the Gauss formula and the precomputed values of the Lagrange polynomials and their derivatives in the basic interval $[0,1]$.

5.5 Examples

In this final section we will solve some boundary value problems with our program and compare the results with those obtained with the program Mathematica. In the first example we solve the equation

$$-\frac{d}{dx}\left(\sqrt{1+x}\ \frac{d}{dx}u(x)\right) + 2u(x) = 50\cos 3x \tag{5.6}$$

```
Sparse *assembleStiffnessMatrix(double a, double b, int intervals, int degree,
                void *a0, void *a1, void *a2, gauss_qdat qdat, Lagrange lv){
  Sparse *K;
  K = xmalloc(sizeof(Sparse));
  triplet_initialize(K, (intervals*degree+2) * 4);

  int n = intervals*degree + 1;
  K->m = K->n = n + 2;

  double xa = a, xb;
  double dx = (b - a)/intervals;
  for(int s=0; s < intervals; s++){//Looping through intervals
    xb = xa + dx;
    for(int i=0; i<=degree; i++){ //Looping through nodes i
      for(int j=i; j<=degree; j++){ //Looping through nodes j
        double kij_sym = get_Kij_symmetrical(xa,xb,i,j,a0,a2,qdat,lv); //K1+K3
        double kij_top = kij_sym + get_Kij_asymmetric(xa,xb,i,j,a1,qdat,lv); //K2 i j
        double kji_bot = kij_sym + get_Kij_asymmetric(xa,xb,j,i,a1,qdat,lv); //K2 j i
        int p = degree*s + i; //Global coordinate of node i
        int q = degree*s + j; //Global coordinate of node j
        if(i == j){
          if(triplet_append(K, p, p, kij_top) == 0)
            return NULL;
        }
        else{
          if(triplet_append(K, p, q, kij_top) == 0)
            return NULL;
          if(triplet_append(K, q, p, kji_bot) == 0)
            return NULL;
        }
      }
    }
    xa = xb;
  }
  if(triplet_append(K,  0 ,  n ,  evaluator_evaluate_x(a2,a)) == 0)
    return NULL;
  if(triplet_append(K, n-1, n+1, -evaluator_evaluate_x(a2,b)) == 0)
    return NULL;
  return K;
```

Source code 5.2: Loops to obtain the stiffness matrix in 1D

in the interval $[0,6]$ with Dirichlet boundary conditions $u(0) = 2.0$ and $u(6) = 1.0$. We write the specifications of the problem in an XML file listed in 5.3

We can see the solution in the figure 5.5. There are two graphics in this figure, in red the solution obtained with Mathematica and in blue the solution obtained with our program.

In the second example, we solve the same equation 5.6, with different boundary conditions. Now we use a Dirichlet boundary condition on the left, $u(0) = 2.0$, and a Neumann boundary condition on the right, $u'(6) = 1.0$. We can see the specification the listing 5.4 and the solution in figure 5.5

In the last example, we will solve the heat equation described in section 2.3. More

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fem-1d>
  <boundaries>
    <!-- A*u' + B*u = C -->
    <left>
      <type>DIRICHLET</type>
      <x> 0.0 </x>
      <C> 2.0 </C>
    </left>
    <right>
      <type>DIRICHLET</type>
      <x> 6.0 </x>
      <C> 1.0 </C>
    </right>
  </boundaries>
  <functions>
    <!-- (a2*u')' + a1*u' + a0*u = f -->
    <a2> sqrt(1+x) </a2>
    <a1> 0 </a1>
    <a0> 2 </a0>
    <f> 50*cos(3*x) </f>
  </functions>
  <solver-settings>
    <quadrature-points> 60 </quadrature-points>
    <lagrange-degree> 4 </lagrange-degree>
    <intervals> 40 </intervals>
  </solver-settings>
</fem-1d>
```

Source code 5.3: First specification of a BVP in 1D

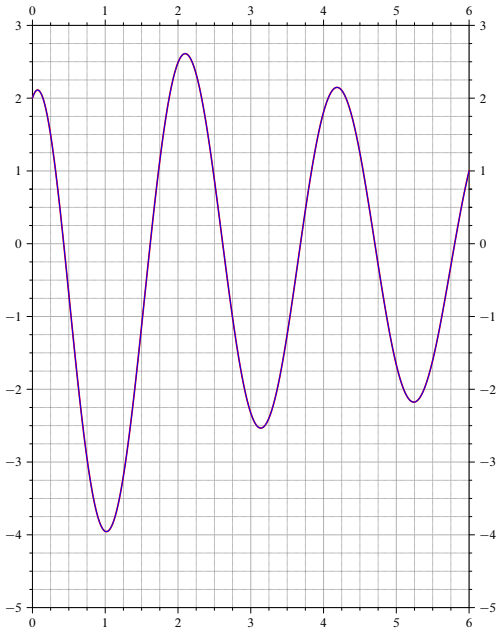

Figure 5.2: Solution of the first BVP example

```
<?xml version="1.0" encoding="UTF-8"?>
<fem-1d>
  <boundaries>
    <!-- A*u' + B*u = C -->
    <left>
      <type>DIRICHLET</type>
      <x> 0.0 </x>
      <C> 2.0 </C>
    </left>
    <right>
      <type>NEUMANN</type>
      <x> 6.0 </x>
      <C> 1.0 </C>
    </right>
  </boundaries>
  <functions>
    <!-- (a2*u')' + a1*u' + a0*u = f -->
    <a2> sqrt(1+x) </a2>
    <a1> 0 </a1>
    <a0> 2 </a0>
    <f> 50*cos(3*x) </f>
  </functions>
  <solver-settings>
    <quadrature-points> 60 </quadrature-points>
    <lagrange-degree> 4 </lagrange-degree>
    <intervals> 30 </intervals>
  </solver-settings>
</fem-1d>
```

Source code 5.4: Second specification of a BVP in 1D

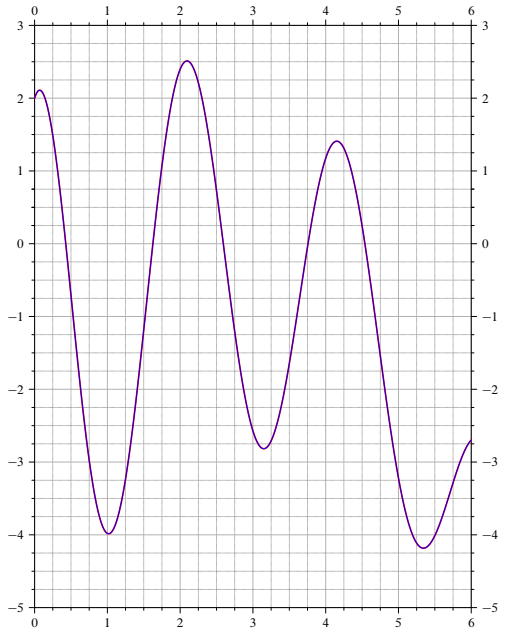

Figure 5.3: Solution of the second BVP example

precisely:

$$\begin{aligned} \Delta u = 0 \qquad & \text{in the interval} \quad [0,10] \\ u(0) = 300 & \\ u(10) = 320 & \end{aligned}$$

In this case the specification file, listed in 5.5, is very simple, and as expected, we obtain a linear function as a solution. See figure 5.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<fem-1d>
  <boundaries>
    <!-- A*u' + B*u = C -->
    <left>
      <type>DIRICHLET</type>
      <x> 0.0 </x>
      <C> 300 </C>
    </left>

    <right>
      <type>DIRICHLET</type>
      <x> 15.0 </x>
      <C> 320 </C>
    </right>

  </boundaries>

  <functions>
    <!-- (a2*u')' + a1*u' + a0*u = f -->
    <a2> 1) </a2>
    <a1> 0 </a1>
    <a0> 0 </a0>
    <f> 0 </f>
  </functions>

  <solver-settings>
    <quadrature-points> 60 </quadrature-points>
    <lagrange-degree> 4 </lagrange-degree>
    <intervals> 30 </intervals>
  </solver-settings>
</fem-1d>
```

Source code 5.5: Specification of a heat problem in 1D

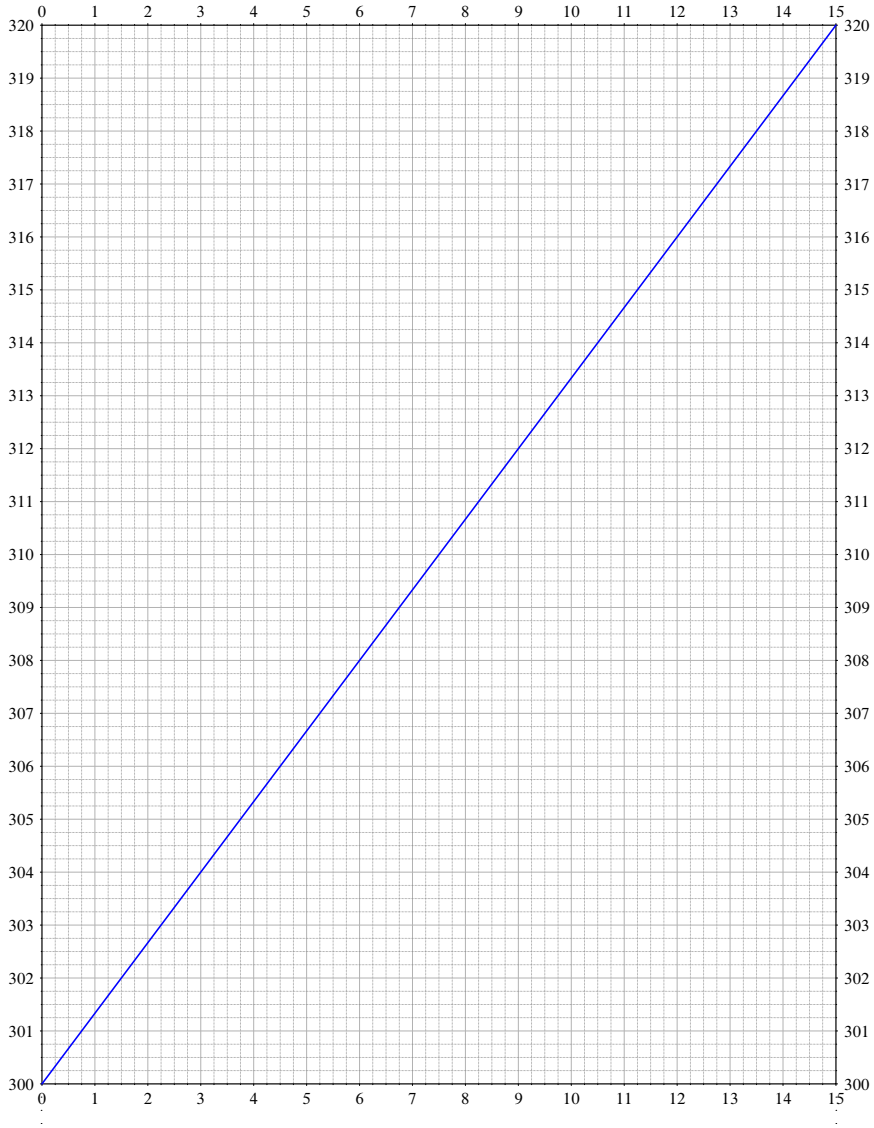

Figure 5.4: Solution to the heat equation

CHAPTER 6

The finite element method in dimension two

6.1 Introduction

All previous chapters have lead to this one. We've learned how to interpolate a function, how to integrate a function on a line and on a triangle. We've also learned how to divide a domain into triangles, and much more. All these tools come together in this chapter to work in unison to solve a PDE on a two-dimensional domain.

6.2 Historical background

We left it off in the previous chapter with the introduction of the term Finite Element by Clough in 1960. Instead of continuing here, we'll go back to the first 2D domain discretizations. In 1943 Courant solved the torsion problem using piecewise linear approximations over a triangular mesh. He had actually partially developed these ideas in the 1920s, but this wasn't discovered until the 1990's. There were multiple papers during the 1950's exploring this idea further. Even in 1953 , McMahon solved a 3D problem with tetrahedral elements.

Although first solutions to simple 2D (and even 3D) problems had been achieved, the method was still lacking in application. The next big challenge researchers in service of the aerospace industry tackled was that of plate bending. It wasn't until the second half of the 60's that these problems where overcome, first by Bazely in 1965 and then by Clough and Johnson in 1968. These methods were only accurate when thin plates had

small deformations, but failed to model plastic deformation. From here on structural engineers started to work on non-linear problems that fall out of the scope of my report.

It took until the 1970's to first apply the method onto simple, linear fluid mechanics, and they very quickly moved on to non-linear time dependent analysis. The first book on the finite element method was written by Brevia in 1978. Even heat and mass transfer was not tackled until 1992. This comes to show that this field is quite a recent one [15].

6.3 Weak form of the problem and the discrete system

The goal of this chapter is the goal of the project: to solve the following partial differential equation:

$$-\nabla(a\nabla u)+B\cdot\nabla u+c\,u=f \tag{6.1}$$

For the sake of brevity and clarity, functions are expressed as f instead of $f(x,y)$. All functions in this chapter are functions of both x and y. $B(x,y)$ is a given vector field $B=(b_0,b_1)$. Hence, a, b_0, b_1, c and f are all given functions of (x,y). The function to solve for is $u(x,y)$.

As with the 1D case, some degree of rigor will be left out of the following development, so as to avoid having to perfectly define the domain, the polynomial space and the solution function. One can read this more technical part in chapters 76 and 77 of [3].

As with the one-dimensional form, we'll now multiply it by all test functions $\varphi(x,y)$ in some test space V' and integrate over the domain Ω.

$$\int_\Omega \big(-\nabla(a\nabla u)+B\cdot\nabla u+c\,u\big)\varphi\;dx\,dy=\int_\Omega f\varphi\;dx\,dy \tag{6.2}$$

If the dimension of the test space V' is finite and the functions $\varphi_0,\varphi_1,\ldots,\varphi_N$ are a basis of this space, then we will have to find a function u that verifies the system of $N+1$ equations

$$\int_\Omega \big(-\nabla(a\nabla u)+B\cdot\nabla u+c\,u\big)\varphi_i\;dx\,dy=\int_\Omega f\varphi_i\;dx\,dy$$

for $i=0,1,\ldots,N$. We can write this equation as

$$\int_\Omega -\nabla(a\nabla u)\varphi_i\;dx\,dy+\int_\Omega B\cdot\nabla u\varphi_i\;dx\,dy+\int_\Omega c\,u\varphi_i\;dx\,dy=\int_\Omega f\varphi_i\;dx\,dy \tag{6.3}$$

and we can apply the divergence theorem to the first integral in the left side. This theorem states that if Ω is a compact region of $\mathbb{R}^2$ with a piecewise smooth boundary

$\Gamma = \partial\Omega$, F is a vector field and g is a function defined in Ω, then

$$-\int_\Omega g\nabla F\, dxdy = \int_\Omega F\nabla g\, dxdy - \int_\Gamma gF\cdot\vec{n}\, ds \tag{6.4}$$

where $\vec{n}$ is the outward pointing unit normal vector of the boundary.

If we apply this theorem to the first integral in 6.3 with $g = \varphi_i$ and $F = a\nabla u$, we obtain that

$$\int_\Omega -\nabla(a\nabla u)\varphi_i\, dxdy = \int_\Omega \big(a\nabla u\big)\cdot\nabla\varphi_i\, dxdy - \int_\Gamma \big(a\nabla u\big)\varphi_i\cdot\vec{n}\, ds$$

and equation 6.3 can be rewritten as

$$\int_\Omega \big(a\nabla u\big)\cdot\nabla\varphi_i\, dxdy + \int_\Omega B\cdot\nabla u\,\varphi_i\, dxdy + \int_\Omega c\, u\,\varphi_i\, dxdy$$
$$-\int_\Gamma \big(a\nabla u\big)\varphi_i\cdot\vec{n}\, ds = \int_\Omega f\varphi_i\, dxdy \tag{6.5}$$

As in dimension one, we reformulate the problem saying that we want to find a function u in a trial space $\hat{V}$ such that 6.5 holds for $i = 0, 1, \ldots, N$. If we take $\hat{V} = V'$, we can write

$$u = \sum_{j=0}^{N} u_j\varphi_j$$
$$\nabla u = \sum_{j=0}^{N} u_j\nabla\varphi_j$$

Now, we have to substitute these values for u and ∇u in the four integrals on the left side of equation 6.5. For the first integral, we have that

$$\int_\Omega \big(a\nabla u\big)\cdot\nabla\varphi_i\, dxdy = \int_\Omega a\left(\sum_{j=0}^{N} u_j\nabla\varphi_j\right)\cdot\nabla\varphi_i\, dxdy$$
$$= \sum_{j=0}^{N} u_j\int_\Omega a\,\nabla\varphi_j\cdot\nabla\varphi_i\, dxdy = \sum_{j=0}^{N} k^1_{i,j}u_j$$

where

$$k^1_{i,j} = \int_\Omega a\,\nabla\varphi_j\cdot\nabla\varphi_i\, dxdy\,.$$

The second integral is

$$\int_\Omega B\cdot\nabla u\,\varphi_i\, dxdy = \int_\Omega B\cdot\left(\sum_{j=0}^{N} u_j\nabla\varphi_j\right)\varphi_i\, dxdy$$
$$= \sum_{j=0}^{N} u_j\int_\Omega \big(B\cdot\nabla\varphi_j\big)\,\varphi_i\, dxdy = \sum_{j=0}^{N} k^2_{i,j}u_j$$

where

$$k^2_{i,j} = \int_\Omega \big(B \cdot \nabla \varphi_j\big)\, \varphi_i \, dx dy \,.$$

The third integral is

$$\begin{aligned}\int_\Omega c\ u\ \varphi_i \, dx dy &= \int_\Omega c \left(\sum_{j=0}^{N} u_j \varphi_j \right) \varphi_i \, dx dy \\ &= \sum_{j=0}^{N} u_j \int_\Omega c\, \varphi_j\, \varphi_i \, dx dy = \sum_{j=0}^{N} k^3_{i,j} u_j\end{aligned}$$

where

$$k^3_{i,j} = \int_\Omega c\, \varphi_j\, \varphi_i \, dx dy \,.$$

The most difficult integral is the last one, because it's a line integral, not a double integral. It can be written as

$$\int_\Gamma \big(a \nabla u\big)\varphi_i \cdot \vec{n}\, ds = \int_\Gamma \varphi_i \big(a \nabla u\big) \cdot \vec{n}\, ds = \int_\Gamma a\, \varphi_i\, \nabla u \cdot \vec{n}\, ds = \int_\Gamma a\, \varphi_i\, \partial_n u\, ds$$

where $\partial_n u$ is the derivative of u in the outward unit normal direction of Γ.

6.4 Computation of K and F

To solve a BVP in dimension two by the Finite Element Method, we have to compute the integrals

$$\begin{aligned} k^1_{i,j} &= \int_\Omega a\, \nabla \varphi_j \cdot \nabla \varphi_i \, dx dy \\ k^2_{i,j} &= \int_\Omega \big(B \cdot \nabla \varphi_j\big)\, \varphi_i \, dx dy \\ k^3_{i,j} &= \int_\Omega c\, \varphi_j\, \varphi_i \, dx dy \\ l_i &= \int_\Gamma a\, \varphi_i\, \partial_n u\, ds \\ f_i &= \int_\Omega f\, \varphi_i \, dx dy \end{aligned}$$

for $i, j = 0, 1, \ldots, N$. First of all, we will explain how many equations and unknowns there are in the final system. As we said in chapter 3, given a triangulation $\mathscr{T}$ of the domain Ω, we consider the space $\mathbb{P}^n_{\mathscr{T}}(\Omega)$ of all continuous functions on Ω such that

their restriction to any triangle $T \in \mathscr{T}$ is a polynomial of degree at most n. Let m be the number of nodes (vertices of the triangles) of the triangulation, e the number of edges and t the number of triangles in $\mathscr{T}$, then the dimension of $\mathbb{P}^n_{\mathscr{T}}(\Omega)$ is

$$N_1 = \dim \mathbb{P}^n_{\mathscr{T}}(\Omega) = m + (n-1)e + \frac{t(n-1)(n-2)}{2}.$$

For example, the dimension of this space in triangulation 4.14 with 6 nodes, 9 edges, 4 triangles and using 5th degree polynomials is 66.

On the other hand, we know that given a triangulation $\mathscr{T}$ of a domain Ω, any integral over this domain can be obtained as

$$\int_\Omega F(x,y)\,dx\,dy = \sum_{T\in\mathscr{T}} \int_T F(x,y)\,dx\,dy\,.$$

From the point of view of a triangle T, only

$$\frac{(n+1)(n+2)}{2}$$

of the functions φ_i and their derivatives are non-zero on triangle T. Then we can loop over the triangles of the triangulation and, for each triangle, compute the values $k^1_{i,j}$, $k^2_{i,j}$, $k^3_{i,j}$ and f_i for the corresponding φ_i and φ_j that are non-zero on this triangle using the formula for Taylor-Wingate-Bos quadrature.

To compute

$$\int_\Gamma a\,\varphi_i\,\partial_n u\,ds\,,$$

we have to take into account that the derivative $\partial_n u$ is also an unknown and that for every edge E in the boundary Γ, we have to compute

$$\int_E a\,\varphi_i\,\partial_n u\,ds\,.$$

In this case, only $n+1$ of the functions φ_i are non null in the edge E so, we have to compute $n+1$ line integrals for every boundary edge. Moreover, E can be easily parametrized. Since it's a straight line:

$$\int_E a\,\varphi_i\,\partial_n u\,ds = \int_0^1 a(\gamma(t))\varphi_i(\gamma(t))\partial_n u(\gamma(t))\gamma'(t)dt$$

and if we approximate $\partial_n u$ with a polynomial of degree n, we have that

$$\partial_n u(\gamma(t)) = \sum_{j=0}^{n} \dot{u}^E_j \mathscr{L}^j_n(t),$$

where $\dot{u}_j^E$ for $j = 0, 1, \ldots, n$ are the unknown values of $\partial_n u$ at the n points of the edge E.

Then, in the equation corresponding the the function φ_i of 6.5, if φ_i is non-zero in an edge E, we have the terms

$$\sum_{j=0}^{n} \ell_{i,j}\, \dot{u}_j^E \qquad \text{where} \qquad \ell_{i,j} = \int_0^1 a(\gamma(t))\varphi_i(\gamma(t))\mathscr{L}_n^j(t)\gamma'(t)dt\,. \tag{6.6}$$

If the triangulation has p vertices and q edges that lie on the boundary, we have

$$N_2 = p + (n-1)q$$

unknowns of the type $\dot{u}_j$, then the total number of unknowns in the discrete system is (figure 4.14 can be helpful again)

$$m + (n-1)e + \frac{t(n-1)(n-2)}{2} + p + (n-1)q\,.$$

Finally, we have a boundary condition for every boundary point, so the number of equations and unknowns in the discrete system is the same. The listing 6.2 contains the basic structure of the computation of the stiffness matrix and the load vector in a two-dimensional boundary value problem solved by the finite element method.

The schema of the system $KU = F$ is similar to that of dimension one. See figure 6.1.

Then, for every boundary edge of the triangle we have to add the therms corresponding to equation 6.6 and the boundary conditions. Listing 6.2 contains the part corresponding to the unknowns $\dot{u}_j$.

6.5 Examples

In the first example, we solve the Poisson equation

$$-\Delta u = \frac{xy}{30}$$

in the unit circle with Dirichlet boundary conditions $u = 0$ on Γ. The description of the domain and the specification of the problem in XML files are listed in 6.3 and 6.4.

We have solved this with our program (see figure 6.5) and with the FreeFem++ package and the solutions obtained are the same.

In the second example, we have solved the Poisson equation

$$-\Delta u = 10\exp(-((x-0.5)^2 + (y-0.5)^2)/0.02)$$

in the unit square with a Dirichlet boundary condition

$$u = 0 \qquad \text{if} \qquad x = 0 \quad \text{or} \quad x = 1$$

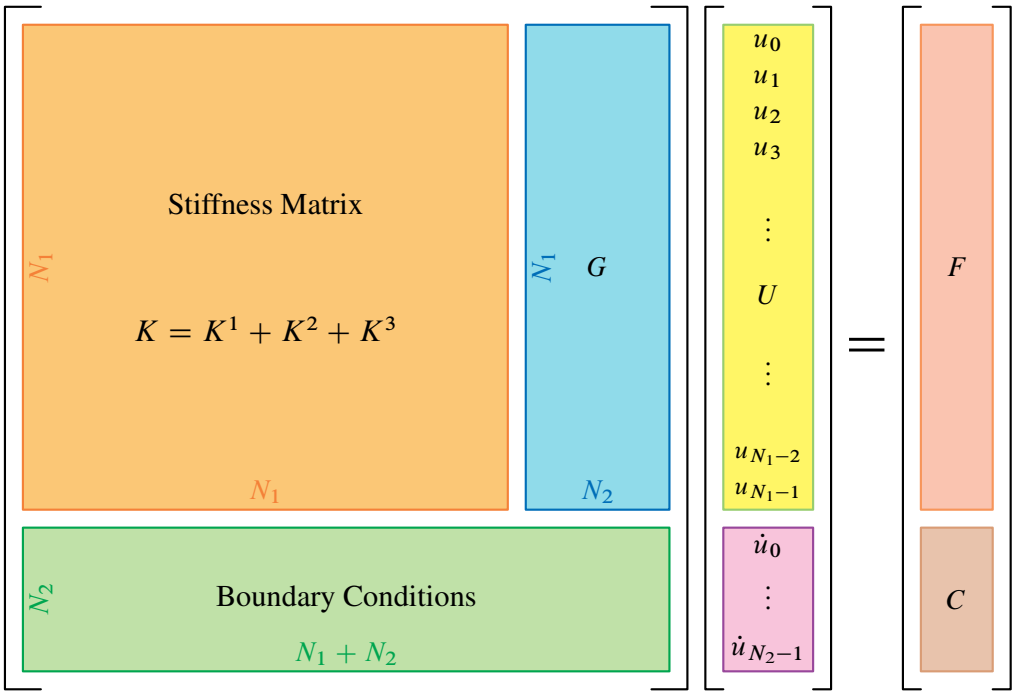

Figure 6.1: Map of the linear system of equations for a 2D domain

```
int assembleSystem(Sparse *K, double *F, int deg, functions *fun, mesh *Mesh,
                gauss_qdat *qdat1d, TWB_qdat *qdat2d, Lagrange *lg1d, Lagrange2D *lg2d){
   //Load Vector
   for(int i=0; i<Mesh->npoints + Mesh->n_boundary_points; i++){
      F[i]=0;
   }

   for(int e=0; e<Mesh->nelems; e++){
      elem *Elem = &(Mesh->elems[e]);
      for(int i=deg; i>=0; i--){
         for(int j=0; j<=deg-i; j++){
            int global = Elem->Points[i][j];
            F[global] += getF(i,j,Elem,fun->f,qdat2d,lg2d);
         }
      }
   }

   //Stiffness Matrix initialization
   K->m = K->n = Mesh->npoints + Mesh->n_boundary_points;
   triplet_initialize(K, 10*deg*deg * K->m); //Guess on number of entries

   //Surface integrals A
   for(int e=0; e<Mesh->nelems; e++){
      elem *Elem = &(Mesh->elems[e]);

      for(int i1=0; i1<=deg; i1++){
         for(int j1=0; j1<=deg-i1; j1++){

            int g1 = Elem->Points[i1][j1]; //Global coordinate of 1st point

            for(int i2=i1; i2<=deg; i2++){
               for(int j2 = (i1==i2) ? j1 : 0; j2<=deg-i2; j2++){

                  int g2 = Elem->Points[i2][j2]; //Global coordinate of 2nd point

                  double K_ij, K_ji;
                  getK(&K_ij, &K_ji, i1, j1, i2, j2, Elem, fun, qdat2d, lg2d);

                  triplet_append(K, g1, g2, K_ij);
                  if(g1 != g2)
                     triplet_append(K, g2, g1, K_ji);
               }
            }
         }
      }
   }
   //Continues in next listing
```

Source code 6.1: Loops to obtain the stiffness matrix and load vector in 2D

```
   //Continuation from previous listing

   //Boundary integrals G and boundary conditions
   for(int e=0; e<Mesh->nedges; e++){
      edge *Edge = &(Mesh->edges[e]);

      if(Edge->bc == FEM_BC_NONE) continue;   //Non-border edges skipped

      for(int i=0; i<deg+1; i++){
         int g = Edge->Points[i];             //global coordinate
         int b = Edge->Points_boundary[i];    //boundary coordinate

         //Boundary integrals G
         for(int j=0; j<deg+1; j++){

            double Gij = getG(i, j, fun->a, Edge->n[0], Edge->n[1], qdat1d, lg1d);
            triplet_append(K, g, Edge->Points_boundary[j], Gij);
         }

         //Boundary conditions (mid-edge)
         if(i!=0 && i !=deg){
            double x,y;
            edgeCoordinates(x,y, Edge, i, deg);
            boundaryConditions(K, F, Edge->bc, x, y, fun, g, b);
         }
      }
   }

   //Boundary conditions (vertices)
   for(int n=0; n<Mesh->nnodes; n++){
      node *N = &(Mesh->nodes[n]);
      boundaryConditions(K,F,N->bc,N->x,N->y,fun, N->nodeno, N->boundaryno);
   }
   return 1;
}
```

Source code 6.2: Terms for the $\dot{u}$ unknowns

```
<?xml version="1.0" encoding="UTF-8"?>
<domain-2d>
  <region type="ellipse" bc="2">
    <center x="0" y="0" ax="1.0" ay="1.0" segments="36" />
  </region>
</domain-2d>
```

Source code 6.3: Specification file of the first example in 2D

```
<?xml version="1.0" encoding="UTF-8"?>

<fem-2d>

   <functions>
      <a exp="1.0"/>
      <b0 exp="0.0"/>
      <b1 exp="0.0"/>
      <b2 exp="0.0"/>
      <c exp="0.0"/>
      <f exp="x*y/30"/>
      <g exp="0.0"/>
   </functions>

   <solver-settings q="100" d="3" a="0.05"/>

   <domain>data/circle.xml</domain>

</fem-2d>
```

Source code 6.4: Specification file of the first example in 2D

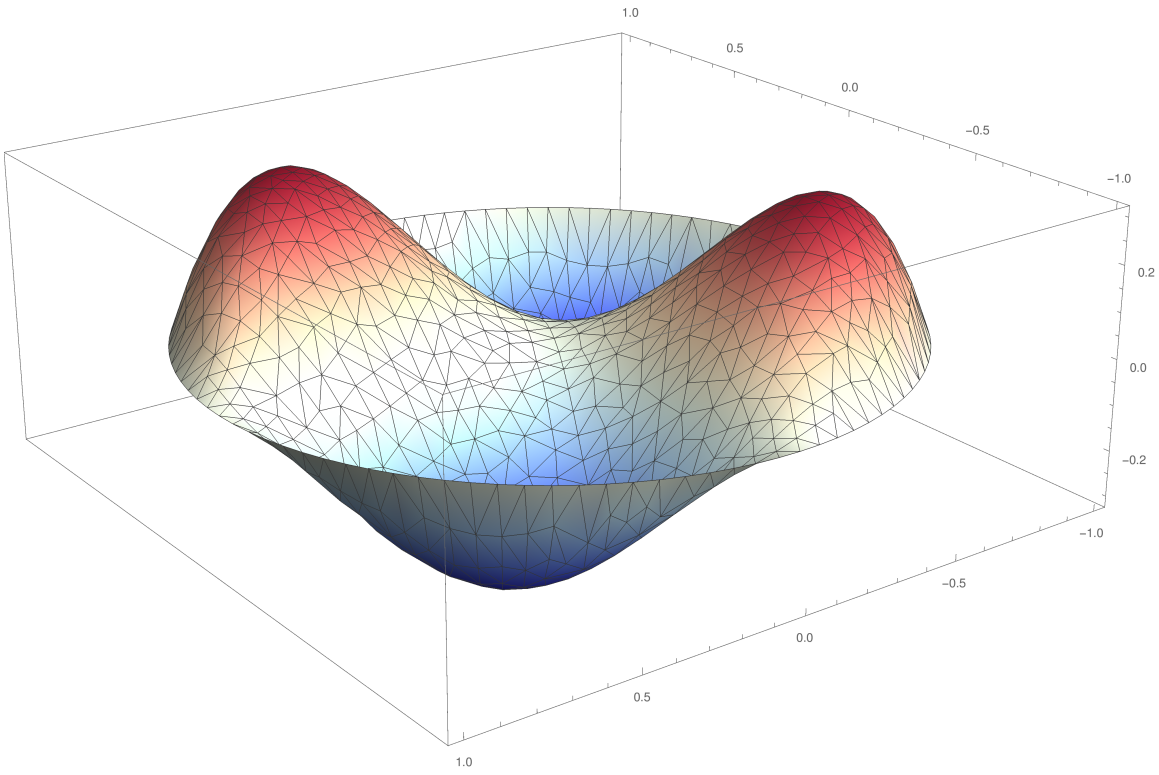

Figure 6.2: Solution of the first BVP in 2D

and the Neumann condition

$$\partial_n u = \cos 5x \qquad \text{if} \qquad y = 0 \quad \text{or} \quad y = 1\,.$$

In all four vertices of the unit square, we apply the Dirichlet condition. We have solved this BVP with the FEniCS package and with our program and we can see the solution in figure 6.3.

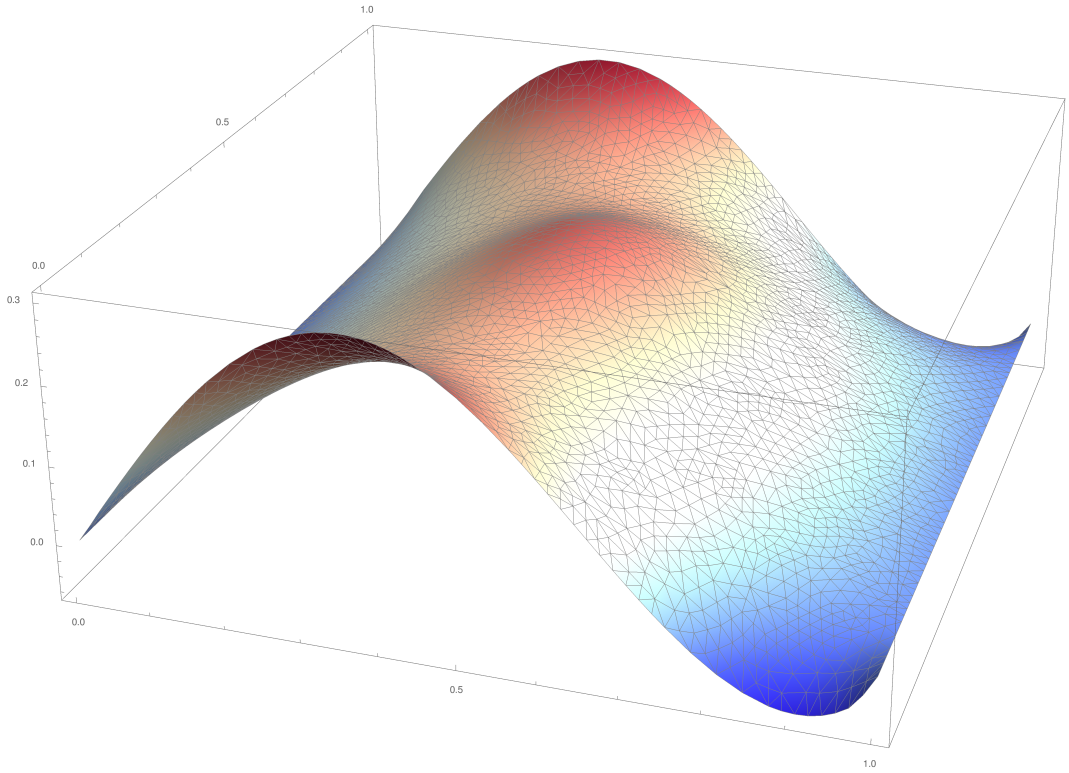

Figure 6.3: Solution of the second BVP in 2D

Finally, we will solve the second problem exposed in section 2.3. As can be seen in equation 2.10. However the result is a bit boring (linear from inner circle to outer section) since once again it follows Laplace's equation. To make it more interesting we need some other term in the equation. To do so we're going to have to use our imagination and suppose there is some material that emits heat in proportion to its temperature:

$$q_{in} = 0.1 + 2u$$

nevermind the logic behind it, the goal is to use the solver to see what would happen. For the other values we will use the heat conduction coefficient of steel and the appropriate heat diffusion coefficients for air around a cylinder and water through a tube.

All this data was obtained from [2]. We set the temperatures at 20°C outside and 80°C inside. The boundary value problem then becomes:

$$\begin{aligned} -\nabla(43\nabla u) - 2u &= 0.1 \quad \text{in } \Omega \\ u + 0.86\frac{\partial u}{\partial n} &= 20 \quad \text{in } \Gamma_o \\ u + 0.086\frac{\partial u}{\partial n} &= 80 \quad \text{in } \Gamma_i \end{aligned}$$

the result can be seen in equation 6.4. Something of note is the fact that despite having the border in contact with a fluid at 20 and 80 Celsius, the border of our solution is not at this temperature (it ranges between 40 and 70 Celsius). This is not because of our magic heat-making material. This happens because we used a Robin boundary condition, and not a Dirichlet one. This emulates the fact that the tube itself would heat up the cold air and cool down the hot water. Since water is more conductive the inner tube's temperature difference with the fluid (70 to 80 Celsius) is lesser that that of the outside boundary (20 to 40 Celsius).

As we saw in equation 2.7, heat transfer is proportional to the gradient of the temperature distribution. As expected, most of the heat takes the short route between the inner tube and the cold outside.

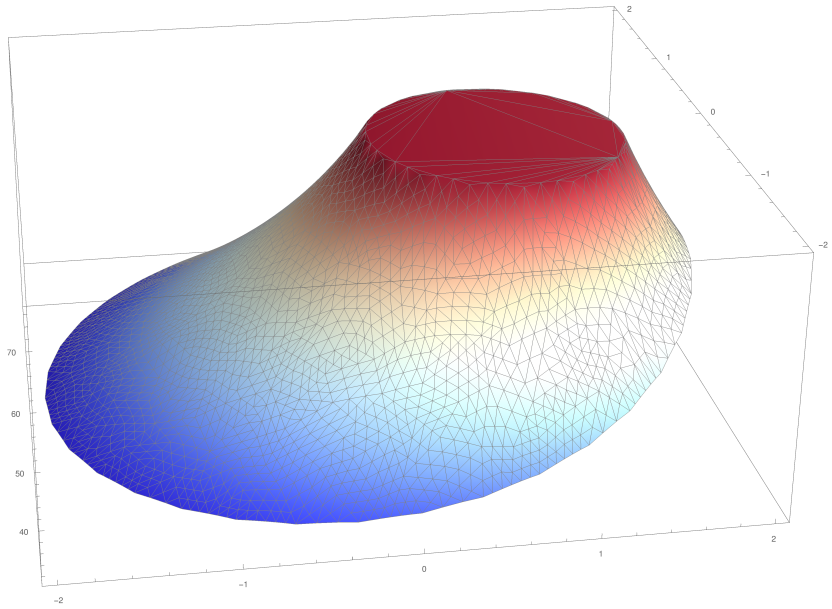

Figure 6.4: Solution of the third BVP in 2D

CHAPTER 7

Conclusion

Although there is no grand conclusion to the project, there are a few things worth mentioning that did not fit in previous chapters. For example, nowhere in the document (so far) is the main file of the project. There are two of them, one for 1D domains and another for 2D domains. The 2D version can be found in listing 7.1. The rest of the code is annexed aside from this document due to its length (about 6000 lines of C code). The main file is not heavily commented but one can follow it by reading the messages displayed in the console (in the `printf` function).

It is clear at first sight that the code is not perfectly polished, it could perhaps be cleaner. Indeed the only thing missing is aesthetics, but that was out of question as soon as I chose to write it in C. Nevertheless, what it doesn't have in beauty it has in use. It can solve all linear differential equations of degree 2 or lower on any arbitrary 2D domain. It then outputs both the mesh and the result of the function in order to use plotting software to draw it (Mathematica, for instance).

Another strength is error handling, some programs can be very obtuse with their errors, instead mine will tell the user where they made their mistake. Few errors might still fall through the cracks but even then rarely does the program crash, instead throwing an error message and closing.

The program can still be improved and that idea is explored later, but that is always true, and one must not let perfect be the enemy of the good. All in all the result is satisfactory, and I dare say better than what what I initially anticipated.

7.1 Further reading

Although all sources in the bibliography were useful to me, some of them were only useful because of a certain section or for some other specific reason. There were a couple of books, however, that were very valuable to me.

If you are interested in other difficult yes attainable programming projects in C, I must refer you to Programming Projects in C, by Rostamian [9]. Not only does he teach you how to implement some very interesting programs, he also encourages a good praxis and in general an approach to programming that helps you stay organized.

On the other hand, if one desires to read more on the topic of the Finite Element Method, and fears not the world of mathematics, I must recommend Understanding and Implementing the Finite Element Method, by Gockenbach [5].

One topic that has been purposefully left out of the scope of the project was the evaluation of the error of the FEM. If you would like to read about it, chapter 4 of [15] goes in depth into this topic. If one is interested in the error in interpolating with polynomials, Ganesan and Tobisca [4] dedicate a whole chapter to it.

7.2 Further work

The program as of now is fully working and it does what it set out to do, however there are certain things it could be improved upon. Here I expose them roughly in order of priority. For starters, it would be relatively easy to implement some sort of user interface to interact with it, and have a program write the xml file automatically.

On the more technical side, it would be convenient to make it support three dimensional domains, hence making it more useful for real-world applications. This, however, would be no small task and could be a final degree thesis on its own.

Also on the technical side, it could be expanded to support vector fields in order to support some idealized fluid mechanics or solve for small elastic deformation and stress distribution. At the moment it can only solve a very idealized version of them since it only solves for scalar fields. In the same vein, it could be made to solve coupled PDE (of which vector field PDE are just a type of). Another possible improvement would be to make it solve for time-dependent equations.

An increase in accuracy and speed could be attained by using an adaptative mesh, that becomes finer or courser in different regions of the domain according to certain parameters.

From a less technical, but more quality-of-life approach, it would be useful to accept standard formats for 2D and 3D modelling, such as STL, ONF, 3DS, etc.

If all these improvements were made, the only missing piece to go commercial would be validation. So far I've tested the result against other solvers, making it, at best, just as accurate as them. I would need some other validation scheme.

```c
#include "fem2d.h"

int main(int argc, char const *argv[]){
   Lagrange lg1d;
   Lagrange2D lg2d;
   gauss_qdat qdat1d;
   TWB_qdat qdat2d;
   Sparse *K;
   double *F, *U;
   mesh *Mesh;
   problem_data pdata;
   lg1d = NULL;
   lg2d = NULL;
   qdat1d = NULL;
   qdat2d = NULL;
   K = NULL;
   F = U = NULL;
   Mesh = NULL;
   pdata.spec = NULL;
   char *filename = NULL;
   if( (filename = readArguments(argc, argv)) == NULL){
      goto end_of_program;
   }
   printf("Reading solver settings and domain...\n");
   if(readXML_2D(filename, &pdata) == 0){
      goto end_of_program;
   }
   printf("Meshing...\n");
   make_vector(Mesh,1);
   if((Mesh = make_mesh(pdata.spec, pdata.meshElementMaxArea, pdata.lgDeg)) == NULL){
      goto end_of_program;
   }
   printf("Loading quadrature data...\n");
   if(load_interpoplation_and_quad_data(&lg1d, &lg2d, &qdat1d, &qdat2d, &pdata) == 0){
      goto end_of_program;
   }
   printf("Assembling system of equations...\n");
   int system_size = Mesh->npoints + Mesh->n_boundary_points;
   make_vector(K,1);
   make_vector(F, system_size);
   if(assembleSystem(K, F, pdata.lgDeg, &pdata.fun, Mesh, &qdat1d,
                     &qdat2d, &lg1d, &lg2d) == 0){
      fprintf(stderr,"There was an issue assembling the system of equations\n");
   }
   printf("Solving system of equations...\n");
   if((U = GSL_solve(K,F,system_size,1e+3,1.0e-6)) == NULL){
      fprintf(stderr,"There was an issue solving the equation system\n");
   }
   goto end_of_program;
   end_of_program: //Memory de-allocation and result printing
      if(U!=NULL){
         storeResult(Mesh,U,";", pdata.lgDeg);
         mathematicaResult(Mesh,U, pdata.lgDeg);
         free_vector(U);
      }
      if(Mesh!=NULL){
         storeMesh(pdata.lgDeg, Mesh);
         storeTriangle(pdata.lgDeg, Mesh);
         free_mesh(Mesh);
      }
      if(lg2d!=NULL){
         free_matrix(lg2d);
      }
      if(K!=NULL){
         triplet_store(K);
         triplet_free(K);
         free(K);
      }
      if(F!=NULL){
         free_vector(F);
      }
   return 0;
}
```

Source code 7.1: Main file for the 2D solver

Bibliography

[1] J. A. DE LOREA, J. RAMBAU, AND F. SANTOS, *Triangulations. Structures for Algorithms and Applications*, Springer, 2010.

[2] ENGINEERS EDGE, *Overall heat transfer coefficient table charts and equation.* `https://www.engineersedge.com/thermodynamics/overall_heat_transfer-table.html`. [Online, accessed: 2019-06-06].

[3] K. ERICSON, D. ESTEP, AND C. JOHNSON, *Applied Mathematics: Body and Soul. Volume 3.*, Springer, 2004.

[4] S. GANESAN AND L. TOBISKA, *Finite Elements. Theory and Algorithms*, Cambridge, 2017.

[5] M. S. GOCKENBACH, *Understanding and Implementing the Finite Element Method*, Siam, 2006.

[6] H. P. LANGTANGEN AND A. LOGG, *Solving PDEs in Python*, Springer, 2017.

[7] A. LOGG, K.-A. MARDAL, G. N. WELLS, ET AL., *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012.

[8] R. ROSTAMIAN, *Programming Projects in C for Students of Engineering, Science, and Mathematics.* `https://userpages.umbc.edu/~rostamia/cbook/`. [Online. Accessed: 2019-03-04].

[9] R. ROSTAMIAN, *Programming Projects in C for Students of Engineering, Science, and Mathematics*, vol. 13, SIAM, 2014.

[10] J. R. SHEWCHUK, *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, M. C. Lin and D. Manocha, eds., vol. 1148 of Lecture Notes in Computer Science, Springer-Verlag, May 1996, pp. 203–222. [Online. Accessed 2019-05-21].

[11] G. STRANG AND G. FIX, *An analysis of the Finite Element Method*, Wellesley Cambridge Press, 2008.

[12] M. A. TAYLOR, B. A. WINGATE, AND L. P. BOS, *Several new quadrature formulas for polynomial integration in the triangle*, arXiv preprint math/0501496, (2007).

[13] D. VEILLARD, *The xml c parser and toolkit of gnome*. http://www.xmlsoft.org/. [Online, accessed: 2019-05-20].

[14] WORLD WIDE WEB CONSORTIUM, *Extensible markup language*. https://www.w3.org/XML/. [Online, accessed: 2019-05-20].

[15] T. I. ZOHDI, *A Finite Element Primer for Beginners*, Springer, 2018.